3-23-2018

Target Detection using Convolutional Neural Networks

Robert P. Loibl

TARGET DETECTION USING CONVOLUTIONAL NEURAL NETWORKS

THESIS

Robert P. Loibl, Captain, USAF

AFIT-ENG-MS-18-M-043

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-043

TARGET DETECTION USING CONVOLUTIONAL NEURAL NETWORKS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Robert P. Loibl, BS

Captain, USAF

March 2018

AFIT-ENG-MS-18-M-043

TARGET DETECTION USING CONVOLUTIONAL NEURAL NETWORKS

Robert P. Loibl, BS

Captain, USAF

Committee Membership:

Dr. Kenneth M. Hopkinson
Chair

Dr. Bryan J. Steward
Member

Dr. Kevin C. Gross
Member

AFIT-ENG-MS-18-M-043

Abstract

This research explores the use of Convolutional Neural Networks (CNNs) to classify targets of interest within satellite imagery. Methods were specifically devised for the classification of airports within Landsat-8 scenes. A novel automated dataset generation technique was developed to create labeled datasets from satellite imagery using only coordinate metadata. Using this approach a very large dataset of over 132,000 labeled images was created without human input. This dataset was used to evaluate the effects of color and resolution on airport classification accuracy. Two experiments were run with the first experiment classifying large airports with 96.8% accuracy, and the second classifying large and medium airports with 90.2% accuracy. Additionally, a new algorithm was developed which optimizes the selection of multi-spectral color bands in order to best trade-off classification accuracy for the number of spectral bands employed.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Kenneth M. Hopkinson, for his guidance and support throughout the course of this thesis effort. Additionally I would like to thank my sponsor, the AFRL Space Vehicles Directorate for both their support and interest in this research.

Robert P. Loibl

Table of Contents

Page

List of Figures

Page

List of Tables

Page

TARGET DETECTION USING CONVOLUTIONAL NEURAL NETWORKS

I. Introduction

Topic & Motivation

The Air Force is facing a new emerging capability gap brought about by the very data its own systems generate. The Air Force, and by extension the Department of Defense (DoD), operate one of the largest and most robust reconnaissance programs in history, gathering and moving massive amounts of data every day. Today turning data into information is largely a human enterprise where intelligence analysts examine Remote Sensing (RS) imagery and fuse multiple data sources to create new information products. Information is then used in turn as the primary tool for military commanders to make decisions. The DoD recognizes the importance of information in military operations and actively pursues Information Superiority against its adversaries [1]. As the Air Force continues to deploy new and more capable sensors to maintain its information advantage, the resources needed to make sense of the incoming data increases. Large amounts of collected data is not used and most data is not exploited to its fullest extent due to manpower limitations. This gap between collected data and produced information threatens the DoD's ability to maintain Information Superiority in the future.

The data to information gap is a serious problem that manifests itself throughout Air Force operations. The speed at which data can be processed and turned into information affects the pace at which decisions can be made. If more data could be processed at a faster rate new operations would become feasible that currently cannot be undertaken. Unfortunately information creation is dependent on limited human resources,

which are unable to deal with the volume of data and the speed at which information needs to be created for new operations. However another option exists to augment and enhance current data processing methods; Artificial Intelligence (AI). AI is a loose collection of algorithmic methods that involve learning, emergent behavior, and advanced problem solving. Specifically this research focuses on the disciplines of machine learning and deep learning.

Automation and AI are already being broadly pursued by the Air Force and the DoD as a whole. To highlight the significance of automation to future Air Force operations "Machine to machine options for turning data into information" is listed as a key capability in the Air Superiority 2030 Flight Plan [2] chartered by the Chief of Staff of the Air Force (CSAF). Human and machine teaming systems also present a near term option to improve the performance of aircraft, Intelligence Surveillance Reconnaissance (ISR), data exploitation, and Communications Command Control (C3). These technologies have the potential of becoming a potent force multiplier for future operations, which is why AI is often cited as a critical component of the so called "Third Offset" [3], allowing whomever masters autonomy to overwhelm their adversaries.

One specific area which is ripe for autonomy is the RS field. Modern AI techniques that have seen success in other commercial domains can be applied to RS data to great effect. Solving the data to information gap for RS imagery using Deep Convolutional Neural Networks (DCNNs) [4] [5] [6] [7] is the focus of this research.

Classifying and localizing objects within images using CNNs is an ongoing area of research [8] [9] [10], and there are specific considerations when applying these techniques to RS data. Several challenges exist that make the RS domain particularly

difficult for machine learning; these include large image size, lack of existing datasets for training, unsupported image formats, proprietary and/or classified data, and small object to image ratios. Despite all of these challenges there are also attributes of this imagery that are beneficial and allow new approaches to be explored.

The first is that almost all satellite imagery contains geographic metadata which can be used in conjunction with object localization to determine a point on earth where the target is located. It can also be used with a known list of object coordinates to create a new labeled dataset. Coordinate information already exists for many objects of interest such as airports, ports, buildings, ships, airplanes, and cars. Cross referencing the locations of these objects within existing satellite scenes of the earth requires a simple search algorithm or database request. Using this method large datasets can be created in an automated fashion. The second benefit is that the coordinates of any detected object can be used as part of a larger system. Automated target detection yielding earth coordinates can be used in a satellite tipping and cueing system. The scenario is as such; one satellite takes an image of the earth, the image is processed using a neural network which yields a potential detection of a target, another satellite is cued to collect further data at the coordinates of the initial target. A third unique attribute is that in addition to typical RGB images, satellites image the earth in various spectral bands [11]. These bands yield additional characteristic information about a target that can increase classification accuracy. Networks trained on multi-spectral image data could potentially analyze an image across dozens of bands simultaneously.

Initial research has begun in the last few years to exploit hyperspectral imagery using CNNs and also to apply CNNs, as well as Recurrent Neural Networks (RNNs), to

satellite imagery. One common approach deals with the lack of labeled hyperspectral training data by training an un-supervised network on generic image data and then applying this network to a small amount of curated and labeled hyper-spectral data [12] [13]. Using unsupervised learning a feature extractor can be created that is generalized enough to also extract salient features from hyper-spectral data. Two self-taught frameworks were tested: the multiscale ICA and the stacked convolutional encoder. These frameworks achieved good results against various existing hyperspectral datasets, but no analysis was done to determine the effect of hyper-spectral data on the achieved accuracy or the extracted features. Research on localizing objects and implementing pixel-wise classification has also been undertaken using hyper-spectral imagery. Using a modified VGG-16 network [5] with its fully connected layers replaced by convolutional layers a hyper-spectral feature map can be created and used for pixel-wise classification [14] [13]. This research also notes the primary disadvantage of pre-training from non-hyper-spectral datasets, which is the reduction in the use of hyper-spectral only features. The end effect of this is an overall reduction of information presented to the network. Using the modified VGG network impressive segmentation maps of the earth were produced, which classified RS imagery by terrain. In addition to the natural use of CNNs to classify RS imagery, RNNs have also been proposed as a feature extractor in hyper-spectral imagery [15]. RNNs are typically used in scenarios where temporal or sequential information is important for accurate classification. When using an RNN for imagery the color bands themselves are fed into the network sequentially. This framework exploits the naturally sequential nature of hyper-spectral image data and represents a significant

departure from other work in the field. Ultimately it was found that RNNs can outperform CNNs when used on imagery with many spectral bands.

Structure of This Paper

Section I continues by discussing the goals of this research and laying out the notional Landsat-8 airport detection problem. Section II covers in detail some of the key AI technologies utilized in this research. In Section III the research methodology is laid out, starting with the automated dataset generation and then the setup and motivation behind each of the two experiments. Section III also explores some of the options considered for two key decisions: the selection of the imagery dataset, and the network architectures for each experiment. Section IV reports the results from both experiments and provides an analysis on the outcomes. Finally Section V provides the conclusions that can be drawn about multi-spectral image classification based on the experimental testing.

Research Goals & Application

This research aims to explore three goals: demonstrate the application of commercial deep learning methods for military missions, understand the specific considerations that the RS domain requires in creating AI systems, and investigate the benefit of multi-spectral information to object detection & classification.

As mentioned before many area of the military can benefit from the introduction of autonomous systems. Two specific military applications that would benefit greatly from this research are satellite tipping & cueing systems and data pre-processing systems for use by intelligence analysts. A data triage system as imagined in this research is a

system in which all incoming remote sensing imagery is fed through. It is a neural network trained to detect a set of military targets within imagery. Such a system can take a "first look" through all data to find the most likely target locations for further investigation by human analysts. These types of human & machine systems are necessary to deal with the increasing volume of incoming data. One way in which a data triage system could take shape is as a heat map of detections overlaid on an RS image. Rather than having an operator search "cold" through the entire image, they can start looking immediately at the "hottest" area of the image. If there are multiple target types each heat map could be saved as a separate channel, allowing the operator to switch focus between targets of interest.

Notional Problem

Research will be conducted by exploring a notional problem to develop tools and methods in an unclassified and simple environment. When deciding on an imagery source and notional target the two main methods of collecting RS data were considered; airborne or space borne collection. The decision was made to focus specifically on RS imagery obtained from satellite payloads. However the methods used could also be applied to imagery obtained via aircraft with slight modifications to data collection and training. The choice of satellite imagery over aircraft imagery is due to several factors, the first being there is better global coverage using satellite imagery as opposed to airborne images. The second factor is that many satellite images have a fixed viewpoint caused by the camera being so far away from the target. This causes images to have a "directly" overhead orientation. When training a neural network differences in orientation make

classification much more difficult. Many airborne camera take images from varying heights, as well as off the side of the plane. This causes the resulting images to be taken from many angles. The expectation then is that more images would be needed or more extensive data augmentation in order to train a network to sufficient accuracy. One drawback to the satellite only approach is that the images are often of a lower spatial resolution, which limits the choice of targets available to train on.

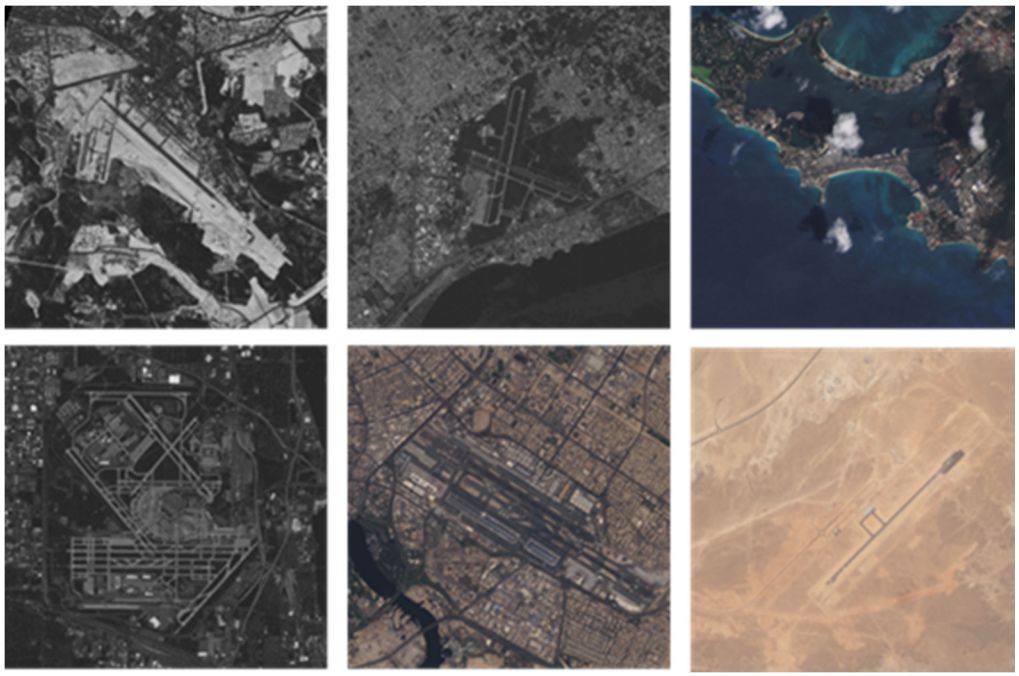

Figure 1. Landsat-8 Images of airports courtesy of the U.S. Geological Survey. Images in the upper left are grayscale, and images in the lower right are composite RGB. All images shown have a spatial resolution of 30m/pixel

Two satellites have been identified as sources of training data for the planned CNNs. The first is the Landsat-8 earth observation satellite [11]. This mission of the Landsat series of satellites is to conduct a long term geological survey of the earth. The satellite carries two sensors, the first is an Operational Land Imager (OLI) and the second is a Thermal Infrared Sensor (TLI). These two sensors together support data acquisition

from 11 different spectral bands. Of the 11 available bands, 7 are of specific interest to my research. These bands are shown in Table 1.

Table 1. Selected Landsat-8 Bands

Band	Wavelength	Resolution
Band 2 – Blue	452–512 nm	30m/pixel
Band 3 – Green	533-590 nm	30m/ pixel
Band 4 – Red	636-673 nm	30m/pixel
Band 5 – Near Infrared (NIR)	851-879 nm	30m/pixel
Band 6 – Short Wave Infrared (SWIR 1)	1566-1651nm	30m/pixel
Band 7 – Short Wave Infrared (SWIR 2)	2107-2294nm	30m/pixel
Band 8 - Panchromatic	503-676 nm	15m/pixel

These bands were selected because they provided images of the earth's surface, rather than aerosols or cloud cover. They also cover the standard RGB colors, as well as additional infrared colors. Finally the panchromatic band allows comparisons between resolution and color to be made. All bands have the same ground resolution of 30 meters per pixel, except for the panchromatic band. This resolution allows for large stationary objects to be used as targets. Some targets that fit this description are cities, military bases, rail stations, ports, and airports. Each Landsat scene covers a broad section of the earth in all bands, the Field of View (FOV) for the payloads is 185 km. The resulting images are about 60mb per image, with the panchromatic images being larger at around 100mb per image. All Landsat data is hosted free of charge on Amazon Web Services (AWS) https://aws.amazon.com/public-datasets/landsat/.

The second source of imagery is from the DigitalGlobe constellation [16] [17]. DigitalGlobe is a commercial company that provides RS imagery at a very high resolution. They currently operate two flagship satellites: Worldview-3, and Worldview-

4. Both satellites collect visible and NIR bands at 1.24m/pixel and panchromatic at 31cm/pixel.

Figure 2. DigitalGlobe Image of Airport (1.24m/pixel)

Table 2. Worldview-3 Bands

WorldView-3		
Band	**Wavelength**	**Resolution**
Blue	445-517 nm	1.24m/pixel
Green	507-586 nm	1.24m/pixel
Red	626-696 nm	1.24m/pixel
NIR	765-899 nm	1.24m/pixel
Panchromatic	450-800 nm	31cm/pixel

Table 3. Worldview-4 Bands

WorldView-4		
Band	**Wavelength**	**Resolution**
Blue	450-510 nm	1.24m/pixel
Green	510-580 nm	1.24m/pixel
Red	655-690 nm	1.24m/pixel
NIR	780-920 nm	1.24m/pixel
Panchromatic	450-800 nm	31cm/pixel

The DigitalGlobe constellation offers the best resolution imagery in the commercial market and opens up many more targets for training. In theory any target resolvable at 31cm/pixel could be used for training, however the following targets have been identified: ships, airplanes, mobile launchers, tanks, and artillery. All of these targets are non-stationary targets that generally relocate about the earth. When compared to the stationary targets that could be trained on with Landsat-8, these targets are much more interesting. Of course because these targets are moving all training examples will need to be found by using a location and a time, which increases the difficulty of acquiring a dataset.

Taking into account the two aforementioned data sources and the potential targets for each It was decided that the research would focus on a notional problem in order to develop generalized tools and methods to train CNNs on RS images.

The imagery source chosen for the primary focus was the Landsat-8 satellite. This was for a couple of reasons: the first is that there is overall more images of the earth available from the Landsat database than DigitalGlobe. This allows for more training examples to be generated for the dataset and improves the odds of success during training. Currently the Landsat-8 database contains over 700,000 scenes of the earth. The second reason is that the size of each Landsat image is much smaller than each DigitalGlobe image. In fact the average size of a Landsat Image is about one tenth the size of a DigitalGlobe image. Since all images must be downloaded from the web, image size has a huge impact on the amount of training examples that can be obtained for the dataset.

The chosen goal is the detection of large and medium airports. Airports were chosen as the target because of their large size, distinctive characteristics, fixed locations, and known geographic coordinates. It was of particular importance that the selected target have known "truth" coordinates so that an automated dataset generation technique could be used. Other moving targets presented a particular challenge when trying to bring together this "truth" information. In future research it may be worthwhile to investigate the acquisition of coordinate information from cars, ships, and planes. Coordinates and earth locations at a certain time could be determined by using combinations of Automatic Identification System (AIS), transponders, and Global Positioning System (GPS) information. However such a task is much harder than simply acquiring a coordinate list of airports around the world and would likely distract from the real research objectives.

II. Background

Perceptron

The perceptron is the most basic building block of a neural network. It is often depicted as a single node with several incoming edges and a single outgoing edge. The design of the perceptron was originally inspired by the biological neuron and is also sometimes referred to as an artificial neuron [18]. Figure 3 shows the layout of a single perceptron, with three input edges and one output edge.

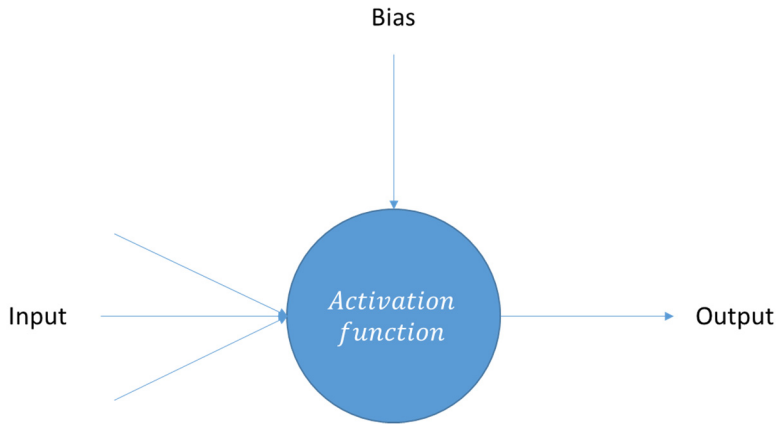

Figure 3. Perceptron

The perceptron functions by first multiplying each input by the associated weight of the incoming edge. Then the product is summed for all of the incoming edges, with the bias added, and then this result is modified through the use of an activation function.

$$output = f\left(\sum_{e \in E} w_e * I_e + b)\right)] \quad (1)$$

Where E is the set of incoming edges, b is the bias, and $f(x)$ is the activation function

Many activation functions exist with the two most popular being the sigmoid and Rectified Linear Unit (ReLU). One major drawback of single perceptrons is that they can only linearly separate classes, and even simple problems such as the XOR function are beyond the capabilities of the perceptron. However networks consisting of multiple perceptrons are able to approximate more complex functions.

Multilayer Perceptrons

More complicated structures can be created using perceptrons as a building block. When multiple perceptrons are combined together in a layered structure with all edges moving in a single direction, a feedforward neural network or Artificial Neural Network (ANN) is created. These networks have a specific arrangement of layers, with the most common being a single input layer, multiple hidden layers, and a single output layer. ANNs are considered to be universal function approximators and can handle very complex problem with the addition of more nodes and layers.

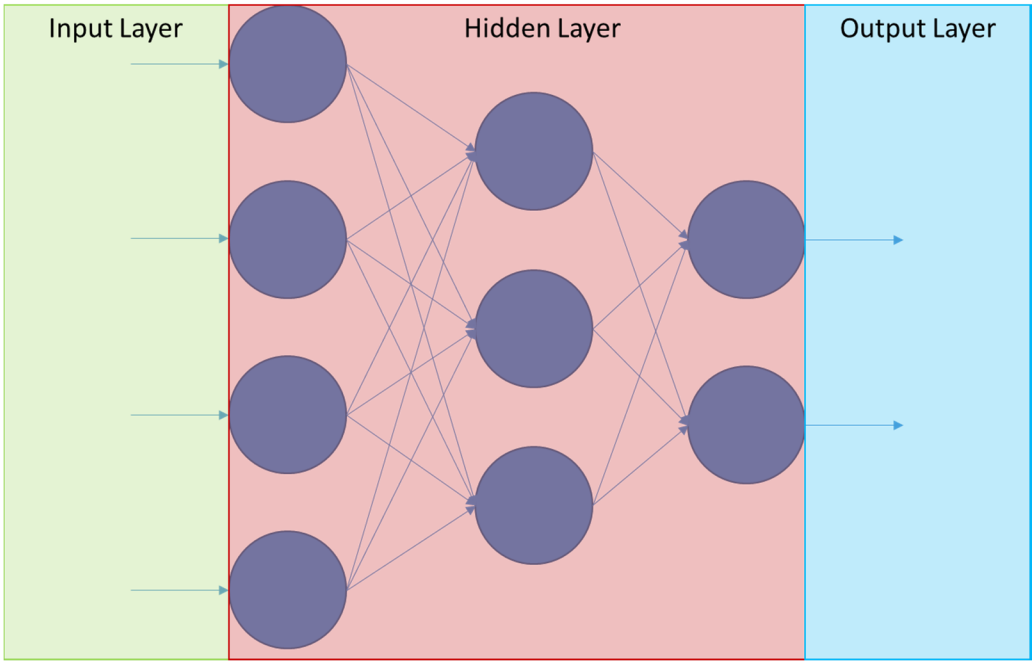

Figure 4. Fully-connected artificial neural network

Each node in Figure 4 represents a perceptron with multiple inputs and multiple outputs. Calculations are performed from left to right with the numerical results flowing from the input to the output. This process can be understood at a macroscopic level as a set of input values being modified by the hidden layer to produce a desired result at the output layer.

Convolutional Neural Networks

CNNs are a special type of ANN which is designed to leverage the spatial relationships of input data. These type of networks are commonly used for image classification, and are currently considered the state of the art approach for this task [7] [19]. CNNs implement several changes to the typical ANN architecture such as the use of the convolution, sparse connections, and weight sharing. The convolution function works by sliding a window across a multi-dimensional array of data. This window is referred to as the convolutional kernel. Each cell of this kernel has a parameter which is multiplied by the corresponding parameter in the input array. The products of each cell in the kernel window are then added together completing the convolution.

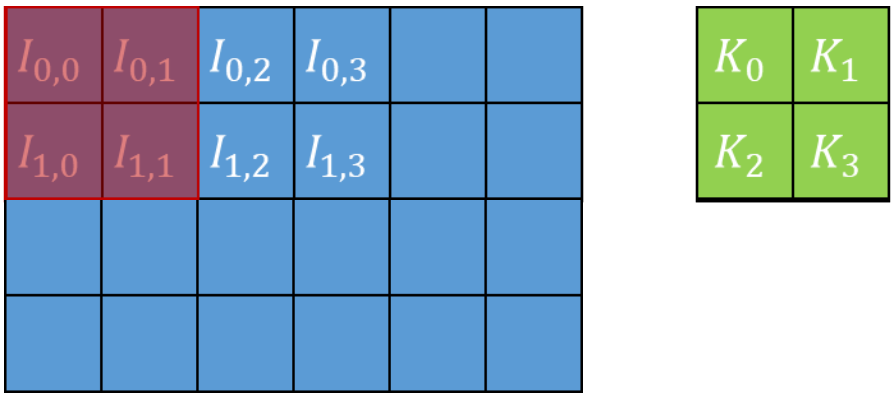

Figure 5. Visual representation of the convolution operation, with the kernel parameters in green, the window in red, and the input matrix in blue.

$$S(i,j) = (I * K) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2}$$

Where I is a two dimensional image, and K is a two dimensional kernel

Since the convolutional function rolls together all of the elements within the kernel window the interaction of these elements is preserved. The convolutional

operation also has several specified hyper-parameters, the first of which is the window size; Figure 5 shows a convolution with a window size of 2x2 elements. The other hyper-parameter is the stride of the convolution which controls how far the window is moved after each convolution, for example if the stride is two then the window will shift two elements over after each calculation. The stride of the convolution affects the size of the output array, as convolutional layers with larger stride need less steps to move across the entire array.

Supervised Learning

When neural networks are trained using labeled training data, it is called supervised learning. When training a network using a supervised approach the goal is to learn a generalized function based on the training dataset. One particular consideration during the supervised learning process is the tradeoff between bias and variance. Bias is error that arises from poor restrictions imposed on the model; as an example when attempting to fit a line to a non-linear function there would be high bias error. Variance is error that arises from a model that is too flexible and sensitive to specific characteristics of the data. The equation below shows that the total expected error of a model is made up of three separate error components: Variance, Bias, and irreducible error ϵ [20].

$$expected\ Test\ MSE = Var\big(f(x)\big) + Bias(f(x))^2 + Var(\epsilon) \quad (3)$$

Furthermore, high bias error causes a supervised network to under-fit the training data and not learn salient features of the data. High variance error causes the supervised network to over-fit the training data and not learn generalizable features. Finding the

proper balance between bias and variance through the use of larger networks and regularization techniques is the key to achieving optimal generalized network performance [21].

Regularization

Regularization is a collection of singular techniques which aim to reduce the amount of overfitting that occurs during the training process. Some commonly used regularization techniques include dropout [22], batch normalization [23], and transfer learning [24]. Each technique has pros and cons, and some techniques like dropout and transfer learning can be used simultaneously. The use of these techniques is particularly important when there is limited training data or a neural network is created with many layers. The number of layers a neural net has is proportional to the amount of capacity it has to approximate the desired function. Capacity however comes at a cost; too much capacity and the network will easily over fit the training data and perform poorly on general data. The use of regularization allows for the use of deeper networks while still preserving the generalized performance of the network. Dropout is the most commonly used of these techniques and it works by selectively disabling connections between nodes during a training epoch. This helps the network by forcing it to not rely too heavily on any given sequence of connections, thus making activations more general and robust. The amount of connections disabled between two layers is a hyper-parameter and values of around 15%-40% are common [22].

Activation Functions

The activation function is an important part of the perceptron as it allows for the modeling of non-linear functions by a neural network. Of the many choices for the activation function the two functions that see widespread use are the sigmoidal function and the ReLU function. Both the sigmoid and ReLU functions are shown in Figure 6.

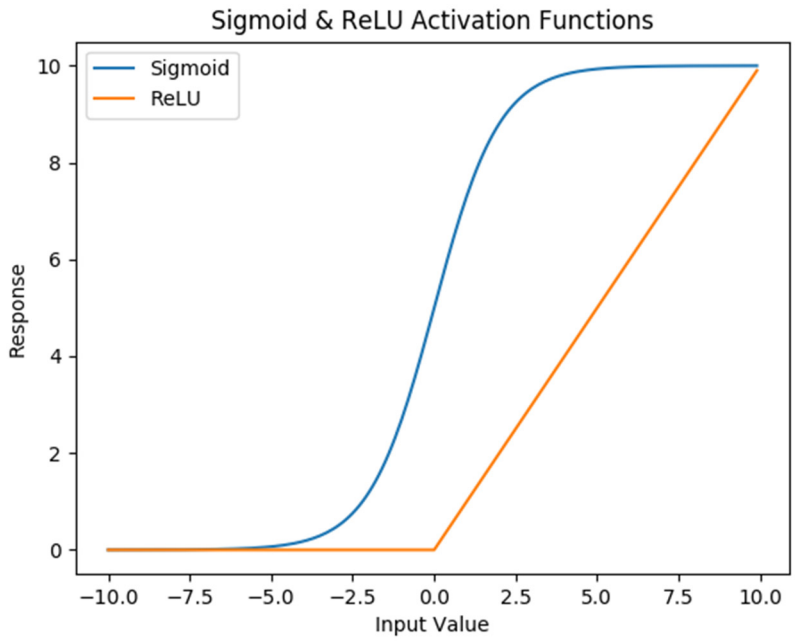

Figure 6. Comparison of commonly used activation functions.

$$sigmoid = \sigma(z) = \frac{1}{1+e^{-z}}, ReLU = R(z) = \max(0, z) \tag{4}$$

Recently the ReLU has overtaken the sigmoid as the activation function of choice for deep neural networks due to its ability to better handle vanishing gradients, its reduced computational complexity, and finally its ability to maintain information through many layers.

III. Methodology

Exploration of this research area was pursued in a progressive fashion that can be broken down into iterative three phases. The first phase is the collection and creation of the dataset from satellite images using automated coordinate based techniques. This results in nearly automated dataset generation which leverages the geospatial metadata on RS imagery. The second phase is a limited exploration of the generated dataset and hardware/software setup, which culminated in the testing of primarily grayscale networks. The testing conducted in the second phase was restricted by software constraints and a limited preliminary dataset. The third phase is where the primary experimentation took place. In this phase all restrictions imposed on the first round of testing were eliminated and a much larger dataset was used. The testing consisted of the selection and training of multi-color networks on a more efficient CNN architecture.

Datasets

Prior to selecting Landsat-8 as a raw imagery source and processing it into a dataset, several pre-existing datasets were investigated for their applicability to this area of research. The following sections describe these datasets and indicates why they were ultimately not suitable for this research. The SAT-4 [25], SAT-6 [25], NLCD 2006 [26], ISPRS Vaihingen [27], ISPRS Potsdam [28], and SpaceNet datasets were all evaluated for use. Finding the right combination of full earth coverage combined with the presence of many multi-spectral bands was challenging and none of the existing datasets fully satisfied the specific needs of the research objectives.

SAT-4 & SAT-6 Datasets

The SAT-4 and SAT-6 datasets consist of data collected from the National Agriculture Imagery Program (NAIP). The imagery contained has four bands; the standard red, green, blue visible bands and one NIR band. The SAT-4 and SAT-6 datasets contain an impressive 500k and 405k cropped image patches respectively. However the labeled classes are general land biomes, with the SAT-4 dataset having 4 categories: barren land, trees, grassland, and all other land. The SAT-6 dataset is similar to the SAT-4 dataset with two new classes added that cover human development: roads and buildings. Both datasets are broken down into 28 x 28 image patches or tiles [25]. Use of this dataset for training an airport detector is not feasible. Several issues arise, the first being that the 28 x 28 tile size doesn't mesh well with most pre-developed ImageNet based networks. This means that no transfer learning can be leveraged when training a network due to the standard ImageNet size being around 256 x 256 pixel tiles. The second issue with this dataset is that the categories don't mesh well with detecting any specific target. While the SAT-6 dataset contains both roads and buildings generally, it doesn't specify any useful notional targets. The third drawback is that this dataset only has 4 bands for use, which would limit the amount of multi-spectral analysis that could be done. The inclusion of a NIR would allow for some limited experimentation to be conducted.

National Land Cover Database

The National Land Cover Database (NLCD) 2006 dataset covers the entire continental United States using Landsat-7 imagery with a pixel resolution of 30m/pixel. Land is classified into one of 16 development and/or Biome categories. While the source imagery has many bands due to its Landsat-7 source [29] it doesn't label structures, but

instead categorizes large tracts of land. Due to the nature of the labeling this dataset was deemed to not be appropriate for training a notional airport target detector.

ISPRS Valihingen & Potsdam Datasets

The International Society for Photogrammetry and Remote Sensing (ISPRS) Valihingen [27] and Potsdam [28] datasets contain semantically labeled imagery. The scope of each dataset is limited to its respective city, with the Vaihingen dataset having 33 patches of labeled data and the Potsdam dataset having 38 patches of labeled data. The images come in several formats consisting of false color images, visible, and visible plus Infrared. In total there are four color bands for every patch. The limited size of these datasets made it a bad fit for global classification of targets. Additionally this dataset was designed to be used when training a semantic segmentation network which assigns a class to each pixel within an image instead of classifying the overall image. This differs from the classification of the entire image and then localization approach that is taken for this research, causing these datasets to not be suitable.

SpaceNet Dataset

The SpaceNet dataset is a relatively new data source and was the last dataset considered for use. SpaceNet was created using DigitalGlobe imagery from the Worldview-3 satellite, and includes truth data for several selected Areas of Interest (AOIs). Currently the following cities are included in the dataset: Rio de Janeiro, Paris, Las Vegas, Shanghai, and Khartoum. Truth data for each city consists of road and building footprints for all cities, and marked Points of Interest (POIs) for Rio de Janeiro. Images are up to 30cm in resolution depending on the sensor band, with up to 8-band multispectral available. While the SpaceNet dataset offers best in class resolution and

multiple spectral bands, it still has limited AOI's for training and limited target types to train a network on. Due to the limited number of AOI's and target types this dataset was ultimately not utilized.

Automated Dataset Generation

After exploring existing public datasets it became clear that all options would present restrictions to the research. Instead the chosen approach was to create a new RS dataset from minimally processed imagery combined with geospatial metadata. The strategy in creating this dataset was to avoid the typical approach of hand labeling a small amount of images. Instead a new approach was devised which leverages coordinates as the source of truth information.

Almost all commercially available satellite imagery includes geospatial metadata which correlates the pixels of the image with a location on the earth's surface. The most common image format which includes this metadata is the GEOTIFF format; geospatial information is contained within headers. From a dataset creation aspect this information presents an opportunity to correlate coordinates with pixels within a given image. Coordinate lists exist for many targets of interest and the pixels of these targets can be found within images using only coordinate information and no further human input.

The pipeline for creating a new dataset using coordinates is shown in Figure 7. This approach was developed using the publicly available Landsat-8 data as an example, but the methodology applies to any collection of GEOTIFF images. The entire process uses only two csv files as an input; the first file is a listing of the target coordinates, and the second file is a listing of the available imagery. The first file contains information about individual targets on each row, including the latitude and longitude coordinates of

each target on the earth. Additional information like the name of a target and other classifying information can also be recorded. The second file contains a listing of all available Landsat-8 scenes in a Bounding Box (BBOX) format. This format indicates the latitude and longitude corner coordinates for each image contained in the listing. Image acquisition time, pre-processing information, cloud cover, and the download URL are also included in the Landsat metadata file.

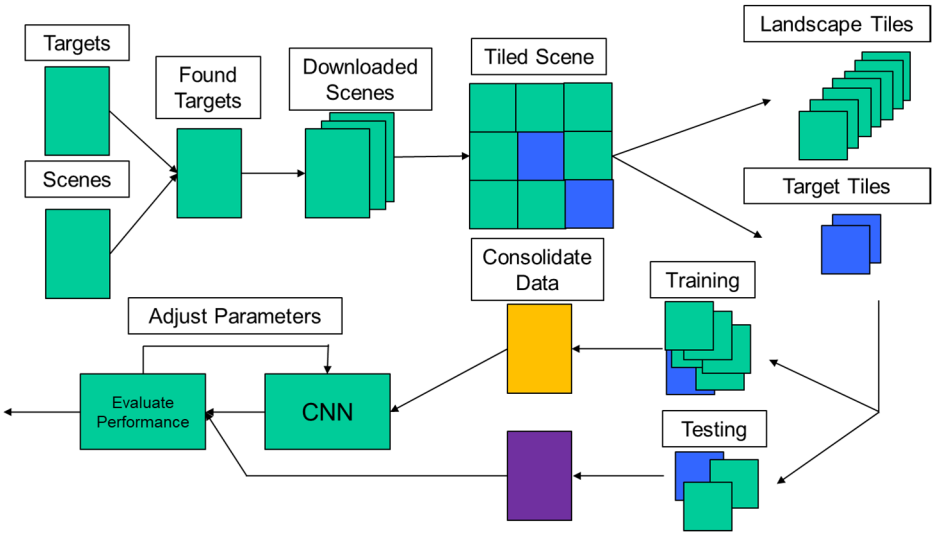

Figure 7. Overview of Automated Dataset Generation Pipeline

The two input files are used in conjunction to perform a search for Landsat scenes which contain targets. The search is flexible enough to handle multiple classes and multiple targets in a single scene. Additionally, when searching, a cloud cover threshold can be specified which will limit the search results to only images with an estimated cloud cover at or below the threshold. Cloud cover estimates are provided in the Landsat metadata file and are created from the Band-9 images. Band 9 is the Cirrus cloud band and the wavelength was chosen to respond well to cloud cover in an image. The output of the search algorithm is a new csv of scenes which contain targets. Information for

multiple targets is concatenated together for use later in the pipeline. An excerpt from the found targets csv is shown below in Figure 8.

name	type	lat	lon
Hewanorra International Airport|Canefield Airport	medium_airport|medium_airport	13.7332|15.3367	-60.9526|-61.3922
Hewanorra International Airport|Canefield Airport	medium_airport|medium_airport	13.7332|15.3367	-60.9526|-61.3922
Piarco International Airport|Guiria Airport	medium_airport|medium_airport	10.5954|10.57408	-61.3372|-62.31267
Piarco International Airport|Guiria Airport	medium_airport|medium_airport	10.5954|10.57408	-61.3372|-62.31267
Guiria Airport|Piarco International Airport	medium_airport|medium_airport	10.57408|10.5954	-62.31267|-61.3372
Piarco International Airport|Guiria Airport	medium_airport|medium_airport	10.5954|10.57408	-61.3372|-62.31267

alt	cloudCover	url
14|13	2.66	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/050/LC80010502015147LGN00/index.html
14|13	4.77	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/050/LC80010502015291LGN00/index.html
58|42	3.94	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/052/LC80010522014288LGN00/index.html
58|42	4.04	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/052/LC80010522015083LGN00/index.html
42|58	2.85	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/052/LC80010522015291LGN00/index.html
58|42	4.37	https://s3-us-west-2.amazonaws.com/landsat-pds/L8/001/052/LC80010522016054LGN00/index.html

Figure 8. Excerpt of rows from found targets CSV

One drawback of the bounding box search method is that a subset of the found scenes don't actually contain targets. This is due to the orientation of Landsat-8 images which are rotated and inscribed within a box. This leads to four triangular areas at the edges of the image which are completely black and contain no information. Removing targets which fall into these areas is solved later in the pipeline with a simple darkness check. Figure 9 shows the erroneous regions described above.

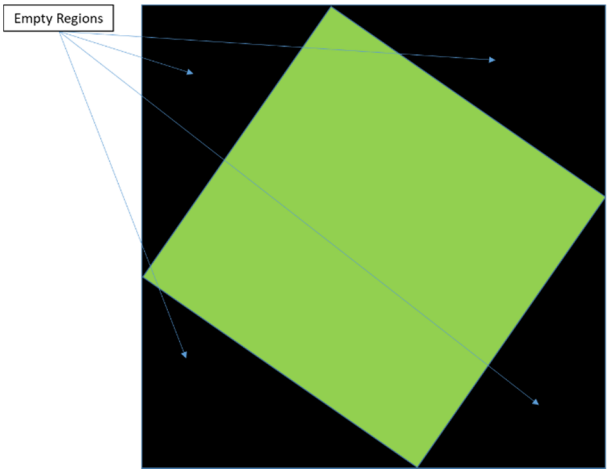

Figure 9. Invalid regions of Landsat-8 scenes

Once the found scene list is generated the next step of the pipeline is to use the provided hyperlinks to download the raw Landsat-8 images. The structure of the hosted database does not support the acquisition of scenes at the scale that AI research necessitates. To resolve this issue a program was written to handle the batch downloading of Landsat-8 scenes. The selection of which bands to download is available through a GUI and the scenes to download are indicated in the found scenes list.

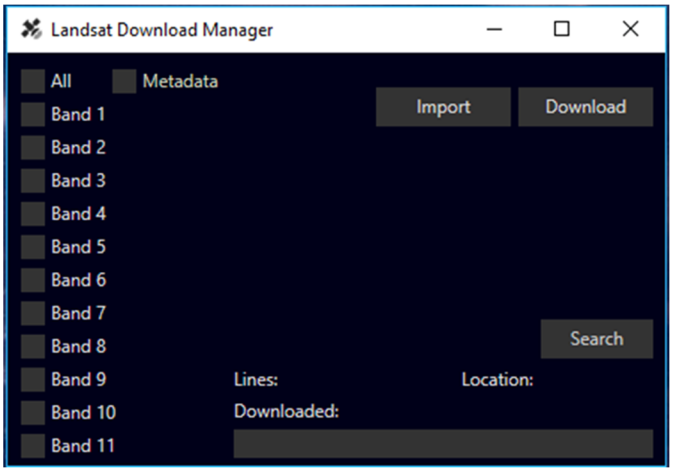

Figure 10. Imagery Download Manager

After the scene downloading phase is over the next step is to turn the raw image into truth labeled tiles. This is accomplished by using the raw Landsat scenes and the original list of target coordinates. The location of any given target is pinpointed within an image by converting the latitude and longitude coordinates into a pixel location. This is accomplished using the Geospatial Data Abstraction Library (GDAL) and results in better precision that a simple percentage offset.

$$m_x = C + p_x * A + p_y * B$$

$$m_y = F + p_x * D + p_y * E \quad (5)$$

$$m_x, m_y = longitude\ and\ latitude\ coordinates$$

$$p_x, p_y = X \ \& \ Y \ pixel \ locations$$

Using the coordinate to pixel conversions in the GDAL library [30], the center-point of the target can be easily found and an area surrounding the center-point can be extracted as a tile. The area of the tile is specified within the program and a nominal size of 256 x 256 was chosen to conform to the historical ImageNet [31] standard. Since the class of each target is annotated in the previous steps, the resulting tile from cropping any given target is also able to be labeled with no further effort. Therefore the tiling algorithm not only creates the dataset images, but also labels them with the truth marking in a single step. Targets themselves are mapped to directly with other parts of the image being tiled as well. The purpose of tiling other portions of the raw image is to create an additional class of background scenery for network training. The background class consists of everything other than the selected targets.

Two different tiling schemes were developed for generating the scenery class tiles. Both approaches share the key features of image and tile boundary preservation. This means that the target and scenery tiles respect the boundaries of the source Landsat-8 image, as well as the scenery tiles additionally respecting the boundaries of the target tiles. This is a very important feature because it prevents the partial inclusion of target tile pixels in other target classes or the scenery tiles. Another shared capability between both schemes is the ability to crop multiple targets and multiple target types simultaneously.

Method one or the “All-Tiling algorithm” attempts to create the maximum amount of tiles from a source image, while still respecting the boundary rules described previously. An average Landsat-8 scene yields between 900 & 1000 tiles, with 1-6 target

tiles and 900+ scenery tiles. The algorithm is shown visually for an example scenario in Figure 11.

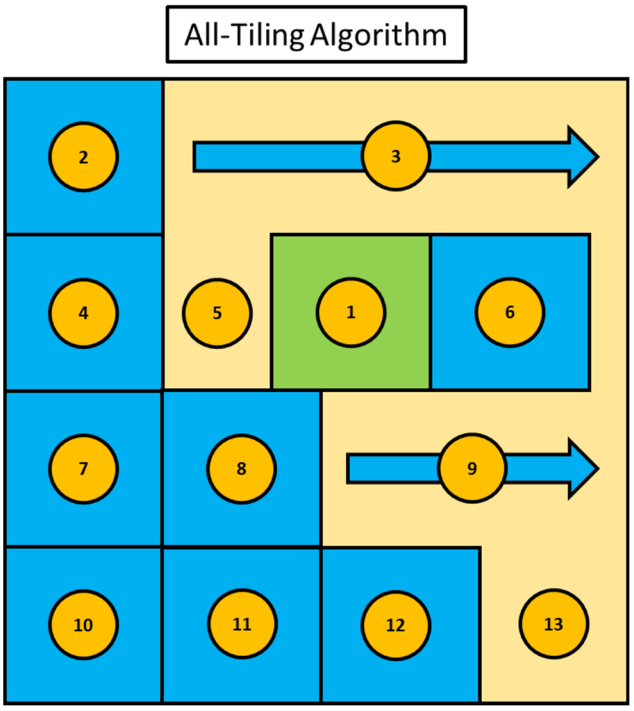

Figure 11. Landsat-8 scene tiled using the All-Tiling Algorithm. Green tiles represent targets & blue tiles represent background scenery

The All-Tiling algorithm starts by first creating tiles for all of the targets in every indicated target class. After generating the target tiles the tile bounds are recorded and referenced during the creation of the scenery tiles. Scenery tile generation starts in the upper left corner of the source image and moves across the image to the right. As tiles are created their bounds are checked against the previously created target tiles and the image bounds. If a candidate tile violates either of these bounds it is not generated and the algorithm skips ahead to the next valid tile. In Figure 11 marker five shows how the algorithm tiles around a target tile, and marker thirteen shows how the algorithm stops creating tiles at the edge of the image.

Method two or the "Adjacent-Tiling algorithm" takes a different approach to tiling the source image. Instead of maximizing the number of generated tiles, it only creates scenery tiles that are touching target tiles. A maximum of 8 scenery tiles are generated for every unique target tile. This leads to a much smaller amount of total tiles generated with the same average of 1-6 target tiles, and a more limited 8-48 scenery tiles.

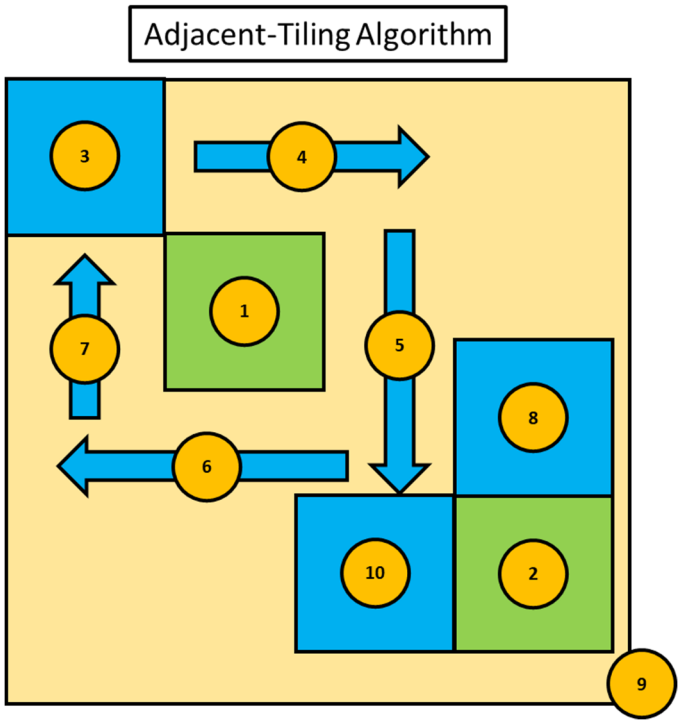

Figure 12. Landsat-8 scene tiled using the Adjacent-Tiling Algorithm. Green tiles represent targets & blue tiles represent background scenery

The Adjacent-Tiling algorithm begins by all of the target tiles first, followed by the generation of the scenery tiles. Following the creation of all target tiles the scenery tile generation start at to the upper left of a given target tile. The first scenery tile is offset by one tile length in the X dimension and one tile length in the Y dimension. The algorithm then proceeds to crop around the target tile in a clockwise fashion until it arrives at the upper left location. Marker's eight, nine, and ten show a situation where

scenery tiles are skipped to avoid duplication and to respect the boundary of the original image.

Enhancements were made to the auto-tiling algorithms to speed up the dataset creation time. Initial estimates projected that the time required to tile all 315,000 potential Landsat images was between 15 and 20 days. While this timeframe was achievable, it represented a huge bottleneck for dataset creation. Parallelization of the auto-tiling code was pursued to dramatically reduce the time required for processing images. This was possible because the tiling of any given image is completely independent, which allows for processing multiple images simultaneously.

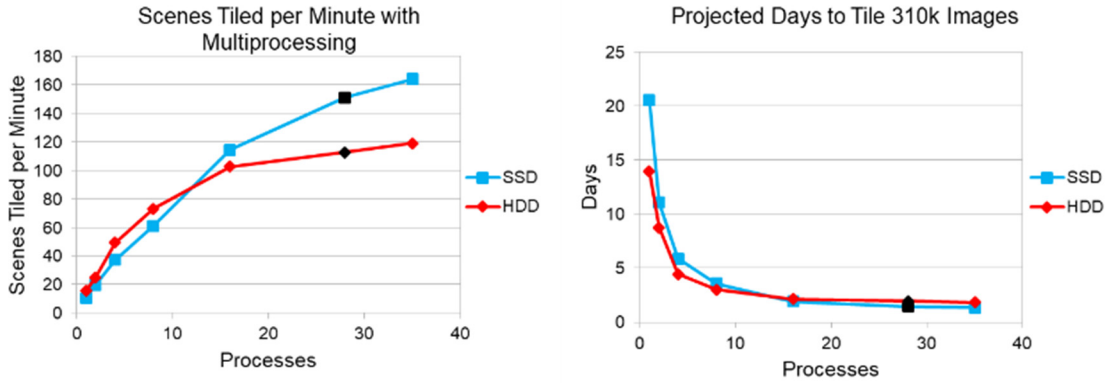

Figure 13. Tiling speedup using multi-processing, darkened points indicate that the number of threads and cores are equal

Table 4. Hardware test configuration for parallel speed characterization

Hardware Configuration	
Operating System	Ubuntu 16.04
GPU #1	Tesla K40
GPU #2	Quadro M6000
RAM	256 GB
SSD	256 GB
HDD	4 TB 7200 RPM
Processor	2 x 14 Core Intel

Figure 13 shows the results of testing the relationship between program speed and the number of CPU cores utilized. Testing was carried out using both HDDs and SSDs, and across 1 to 40 threads. Dark points on the graph indicate where the number of CPU cores equals the number of threads. Optimal performance also occurs at this point, which for the test system is 28 cores. HDD performance is better when the number of threads is less than 12, after which point the SSD begins to perform faster. The fastest throughput recorded for Landsat images was 160 scenes/minute, generating approximately 160k tiles/minute. This translates into nearly 360 hours saved when tiling the entire Landsat dataset.

The output of the auto-tiling algorithms are tiles separated by their class labels. At this stage the total collection of tiles can be split up into training, validation, and test subsets as needed. When generating datasets particular consideration was taken to address the imbalance of tiles between classes. Class imbalance arises from the behavior of the backpropagation algorithm, which rewards or punishes network weights based on the batch classification accuracy during training. If you have many examples of a single class then there will be a proportional amount of weights updated based on this class, which causes the network to perform better on that class. In a multi-class problem, maximizing the classification accuracy of a single class may lead to lower accuracy on all classes with less training examples and ultimately lower overall classification accuracy. Depending on which tiling algorithm is used, there can be over a 100:1 ratio between target and scenery tiles. Two options were considered to deal with the imbalance [32]: adjust the backpropagation algorithm during training or sample from the scenery tiles.

One method of dealing with class imbalance is to normalize the effect that tiles from each class have on the backpropagation algorithm. For the situation described previously this would consist of reducing the effect that scenery tiles have during training to compensate for the increased number of these tiles. The alternative method is to randomly sample from all of the available scenery tiles, which ensures that there are an equal number of target and scenery tiles in the dataset.

Finally the last step of the dataset generation pipeline is to create a dictionary listing with the tile locations in memory and the class labels. This dictionary file is used during the training process to generate batches of tiles for the neural network.

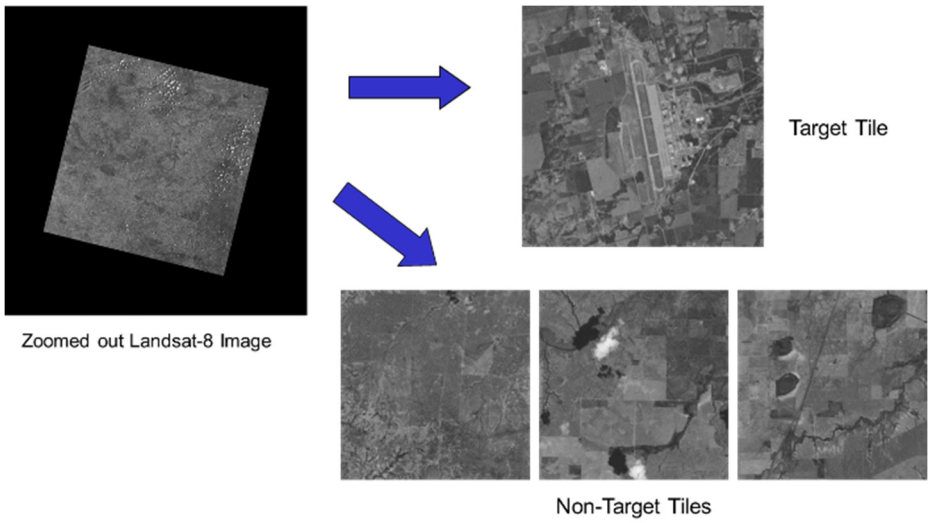

Figure 14. Sample results from Auto-Tiling algorithm. A full size Landsat-8 image is shown in top left, with four resulting tiles shown on the right. A typical Landsat-8 scene yields over 900 individual tiles. Images courtesy of the U.S. Geological Survey.

First Experiment Methodology

For the experimental portion of this research two separate experiments were planned. The first experiment was conducted largely as a proof of concept, with the goal

of validating the notional target selection, imagery source, and automated dataset generation. Several restrictions were placed on this experiment because at the time only a limited dataset was available and default imaging libraries were used for handling data batches for training. Large airports were the only target class considered in this experiment, which led to the number of images in the dataset to be limited. The experiment was also restricted to using networks trained on either single color images or composite RGB images. Finally the color bands were also restricted to only bands with the same pixel resolution of 30m/px (B2-B7). The panchromatic band was the only band not considered since it has double the pixel resolution of all other bands.

Table 5. Dataset for experiment #1 broken down by selected processing stages

Experiment #1 Dataset	
CSV Files	567 Large Airports
Found Airports	2500 scenes x 6 Bands
Generated Tiles	900 – 1000 per scene
Airport Tiles	1007 x 6 Bands
Non Airport Tiles	1007 x 6 Bands
Total Tiles	12,084 Tiles

The dataset used for experiment one is shown above in Table 5, starting with a list of 567 large airports 1007 tiles were generated. An additional 1007 non-airports or scenery tiles were also sampled from the available pool of tiles. The same 2014 target and scenery tiles were collected across B2-B7, resulting in a total of 12,084 tiles for the entire dataset. The total dataset was split into smaller training and testing subsets. A validation set was not created for this experiment because no hyper-parameter tuning or early stopping was used. Preservation of the test set data was maintained by not applying any tuning decisions to it as well. Training and testing size was determined by using a 70%

training and 30% testing split, resulting in a training dataset of 8,388 tiles and a test dataset of 3,696 tiles.

Several network architectures were explored for the first CNN with the ultimate choice being the VGG-19 [5] architecture. The following network architectures were considered: AlexNet [4], VGG-16, VGG-19, ResNet [34], GoogleNet [33], and SqueezeNet [6]. Most of these architecture were developed in response to the ImageNet challenge (ILSVRC) [31], and were at the time of their introduction leading architectures in terms of Top-1 and Top-5 accuracy in the 1000 class problem. Each architecture also leverages the spatial relationships of pixels using convolutional layers in some fashion, leading them to all be considered types of CNNs.

AlexNet

AlexNet was the CNN that started the modern deep learning revolution and at its time it shattered the existing record in the ILSVRC competition. The network introduced a number of key features which when utilized together led to the record breaking accuracy. The architecture consisted of 5 convolutional layers followed by 3 fully connected layers and then finally a 1000-class softmax layer for the output. The network also used Rectified Linear Unit's (ReLUs) for the activation function, Dropout layers to increase regularization, and optimized GPU training. AlexNet was the best performing network in the ILSVRC-2012 competition with a Top-5 test accuracy of 15.3%.

VGG-16 & VGG-19

The Visual Geometry Group (VGG) 16 & 19 architectures represent the natural progression of deep CNNs inspired by AlexNet. The architecture was developed by the Visual Geometry Group at the University of Oxford in 2014. It consists of a deeper

architecture than previous ILSVRC submissions with 16 layers and 19 layers respectively. Increased depth was made possible by reducing the size of the convolutional filters from 11x11 or 7x7 down to 3x3 pixels. The stride of the window is also reduced from either a 4 pixel or 2 pixel stride to the smallest possible 1 pixel stride. VGG architectures also utilize the ReLU activation function for its computational efficiency. With this deeper architecture a very high Top-5 test accuracy of 7.3% was achieved in the ILSVRC-2014 competition. The VGG architecture still sees common use due to its intuitive design and respectable performance.

GoogleNet

The GoogLeNet architecture represents one of the first departures from the paradigm of simply adding more layers. It introduces an entirely new type of architecture building block called an inception module. Through the use of these inceptions modules the architecture is able to increase the width and depth of the network without increases the overall computational load. The final network design employs 22 layers, uses ReLU activation, and has 12x less parameters than AlexNet while also having increased performance. The inception modules themselves consist of a collection of possible layer types including 1x1 conv, 3x3 conv, 5x5 conv, and max pooling. During training the selection of layer type is not pre-defined allowing the network to optimize for the proper inception module subset. Designed with computational efficiency in mind the architecture still logged a Top-5 test error rate of 6.67% in the ILSVRC-2014 competition.

ResNet-152

ResNet-152 is a network architecture developed by Microsoft research for the ILSVRC-2015 competition. It features an incredible 152 layer architecture, which was and still is vastly deeper than any other architecture proposed. Despite the increased number of layers, the complexity of the network is actual lower than AlexNet or VGG. One major innovation of this network is the addition of skip connections from lower layers to deeper layers to combat the vanishing gradient problem [35]. An ensemble of ResNet-152 networks achieved a Top-5 test error rate of 3.57% in the ILSVRC-2015 competition.

SqueezeNet

Finally the last network architecture considered was the SqueezeNet architecture. SqueezeNet aims to reduce the size of the overall network rather than pursue pure performance. This sets it apart from many of the other architectures already discussed and causes the network to have many advantages that make it an attractive choice. Some of the advantages of SqueezeNet are reduced server communication during training, less bandwidth to move trained networks to a deployed system, and the feasibility to deploy the network on memory limited devices like FPGAs. Size reductions are achieved through the use of the fire module, which uses a squeeze and expand design. The squeeze is achieved by using a simple 1x1 convolution layer and the expansion is achieved by following up with both a 1x1 convolution and 3x3 convolution in parallel. These outputs are concatenated and passed to the next layer. The magnitude of size reductions is impressive with the SqueezeNet architecture maintaining AlexNet accuracy with 50x less

parameters. Further reductions in size of up to 510x can be achieved using model compression techniques.

Custom VGG-19 Network

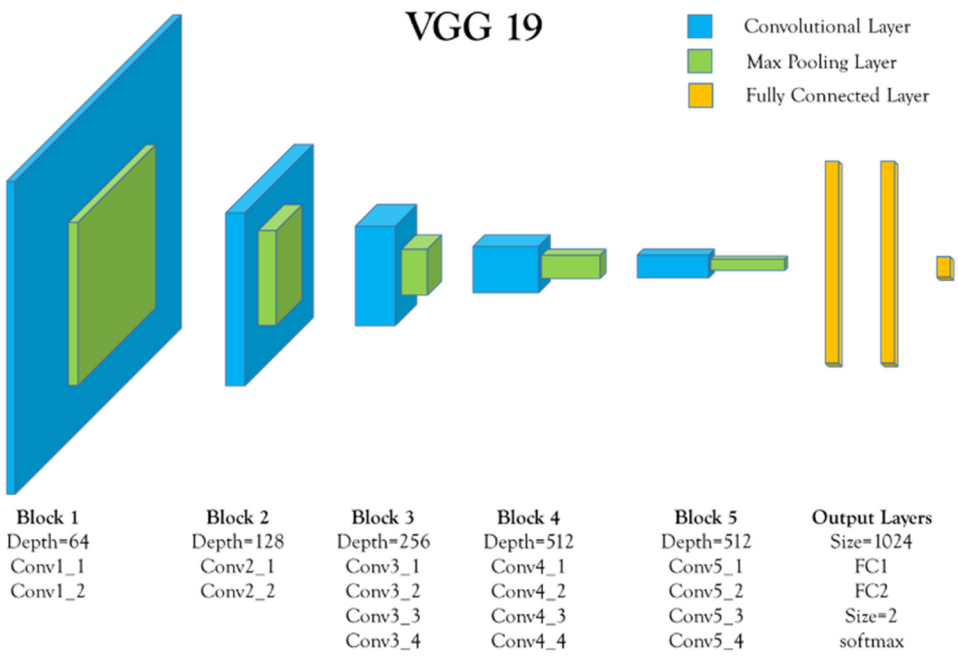

Figure 15. Modified VGG-19 architecture used in experiment one.

After considering all of the network architectures discussed above, a modified version of the VGG-19 [5] architecture was chosen. VGG-19 was chosen because of its simple design and high accuracy. The modifications made to the network are shown in Figure 15; all convolutional and max pooling layers were preserved with the fully connected layer modified to better accommodate a 2-class problem. The fully connected layers were reduced in width from 2048 to 1024 nodes each, with the output layer reduced to a 2-class softmax.

For all tests the modified VGG-19 network was pre-trained using ImageNet weights. Transfer learning was employed for this experiment for a few reasons, the first

reason was that the limited dataset could benefit from the much larger ImageNet dataset. The second reason was that since only grayscale and 3-color RGB networks were being tested, it opened up the possibility of ImageNet weights. If networks were tested using more or less than 3-colors there would be no way to correlate the ImageNet weights, which were developed using only RGB images. The pre-trained VGG-19 architecture's input layer contains one weight value for each pixel and each color, this results in 3 different weights for every pixel representing the pathway from red, green, and blue. Combinations of these weights between colors and between pixels occurs in subsequent convolutional layers. When feeding new data through this pre-trained network the mapping of training image colors to existing color pathways needs to be maintained and the number of colors must be the same. If the number of channels in the input images changes then there would be no corresponding pathway in the network. To address this issue for single band or grayscale networks and benefit from ImageNet pre-training, the single colors were duplicated into all 3 color channels to create a false RGB image as shown in Figure 16 below.

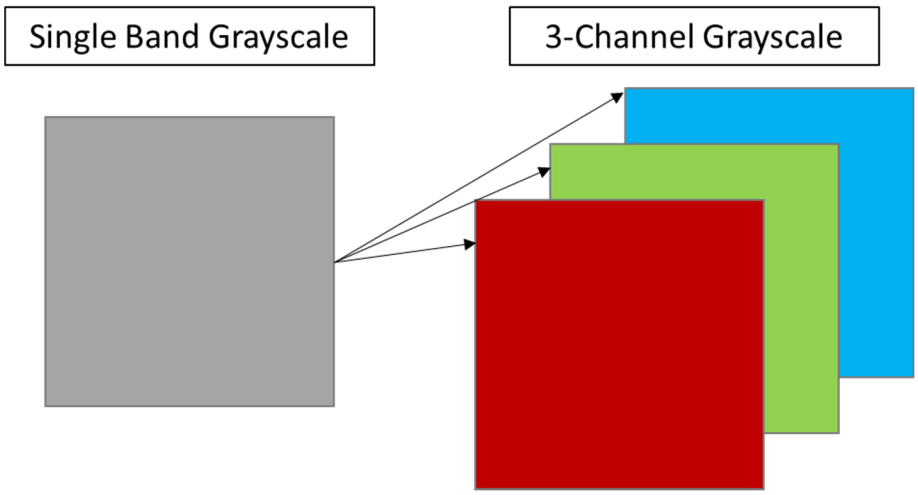

Figure 16. False color images were created by duplicating pixels from a single band into the RGB color channels.

During the network training portion of the experiment certain layers of the network had their weight locked. The only layers that were allowed to be updated by the backpropagation algorithm were the fully connected layers. This meant that the feature extractions being outputted from the convolutional layers were unchanged from the ImageNet dataset. This decision was made again due to the limited amount of data available for the first test. The rationale was that limited data could not have a meaningful effect on so many convolutional layers and could also overwhelm the weights already present from the ImageNet pre-training.

Second Experiment Methodology

The primary goal of the next experiment was to remove all of the restrictions imposed in the first experiment. The removal of those restrictions allowed this experiment to answer several questions simultaneously. Next, since the Panchromatic Band-8 is twice the resolution of the other color bands, the effect of resolution on classification accuracy is also addressed. Another area of investigation was the strategy and benefit of combining many color bands to increase classification accuracy. Finally the experiment explored the use of an alternative architecture which has greater parameter efficiency.

Table 6. Dataset for experiment #2 broken down by selected processing stages.

Experiment #2 Dataset	
CSV Files	5099 Large/Medium Airports
Found Airports	9000 scenes x 7 Bands
Generated Tiles	900 – 1000 per scene
Airport Tiles	9459 x 7 Bands

Non Airport Tiles	9459 x 7 Bands
Total Tiles	132,426 Tiles

The dataset for experiment two is shown above in Table 6. This experiment expands the number of starting targets by including both large and medium airports. Both types of airports are consolidated into a single airport class, with the other class once again being the background scenery. Included in the dataset are tiles from Landsat-8 bands 2 through 8. Bands 2 through 7 have a pixel resolution of 30 meters and Band-8 has a resolution of 15 meters. Broken down by band the dataset contains 9,459 tiles in each class and over 130k tiles across all bands. The dataset was also split further in three subsets: training, validation, and test. The split between the three subsets was 70 percent training, 10 percent validation, and 20 percent test.

The training set contained 6,621 tiles from both the airport and scenery classes. This subset is the only data that the network is shown during the training and backpropagation phase. The validation set consisted of 945 tiles per class, and is used for several different purposed during this experiment. The primary use of the validation set is for hyper-parameter tuning, such as selecting an appropriate learning rate and optimizer. It is also used for selecting an optimal stopping point during the training phase. Finally the validation set was also used for evaluating the similarity of the tiles in each color band. The last subset created is the test set which is used after training to determine the classification accuracy of the network.

For this experiment a new architecture was chosen with fewer parameters than the first experiment. The modified VGG-19 architecture was abandoned for two reasons; the first was that the network was originally designed for use on 1000-class problems and

this research problem includes only two classes. This huge reduction in classes should lead to a reduction in the complexity of features needed to discriminate between targets and scenery. A complexity reduction should furthermore translate into a reduction in network capacity. The second reason for utilizing SqueezeNet is the speed at which new models can be created and trained. From a random weight initialization SqueezeNet models were trained for 200 epochs in about one hour. This benefited the second experiment by allowing multiple models to be trained and evaluated for each color in the same time that one VGG-19 model could be trained. Due to the large amount of samples from each color, statistical methods such as t-tests parzen windows were employed to evaluate the relative significance of results. In light of the two aforementioned reasons the SqueezeNet architecture was utilized. The design of this architecture is shown below in Figures 17 and 18.

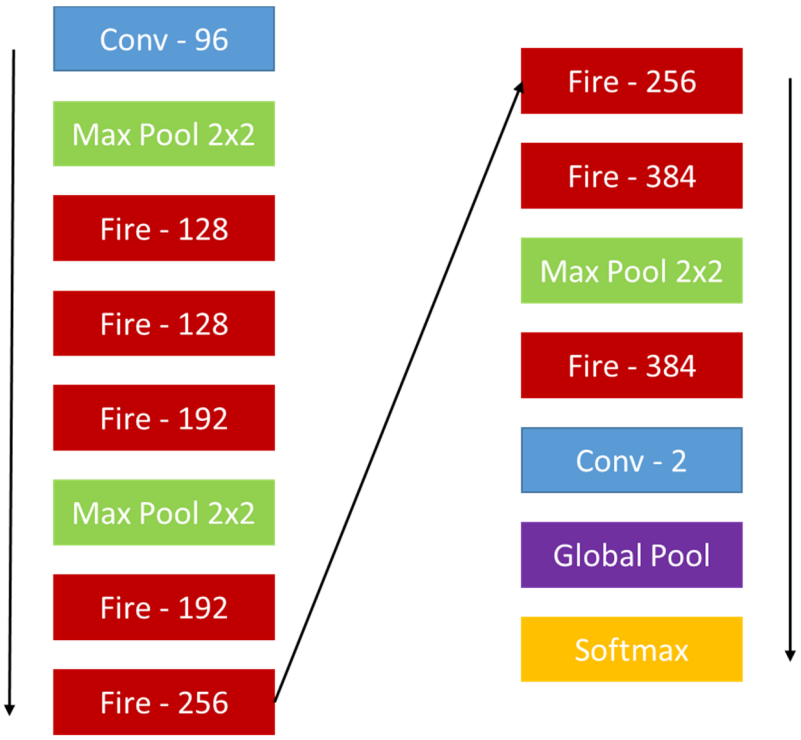

Figure 17. Architecture of SqueezeNet used in experiment two

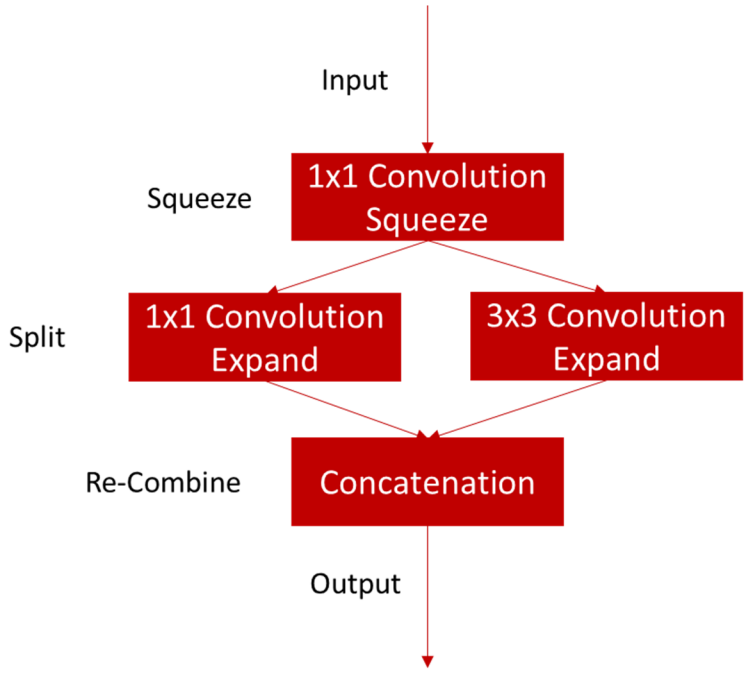

Figure 18. Detailed view of Fire Module

Data collection started with the training of single band networks for all colors. Several networks were trained for each color and the mean classification accuracies were recorded for each network. In the course of training, some networks remain stuck at a local minimum of around 50% classification accuracy. This situation is caused by the initial weights that are randomly assigned prior to network training. When this happens the training for the network is cut short and the results are ignored when calculating the mean classification accuracies.

Following the single color network testing the experiment proceeded to multi-color networks. These networks were trained on images with anywhere from 2 to 6 color channels. The selection of which colors to combine and in what order to combine them is an open research question. Two different methods were tested during the course of the

experiment. Both methods operate by adding colors greedily based on a calculated metric. This approach is a modified case of the more general set cover problem [35], where the goal of the algorithm is to find a collection of subsets whose union covers the entire desired set.

$$
\begin{aligned}
&Greedy\ Set\ Cover:\\
&Start\ with\ R = U\ and\ no\ sets\ selected\\
&While\ R\ \neq \emptyset\\
&Select\ set\ S_i\ that\ minimizes\ {}^{w_i}/_{|S_i \cap R|}\\
&Delete\ set\ S_i\ from\ R\\
&EndWhile\\
&Return\ to\ the\ selected\ sets
\end{aligned}
$$

In this case the desired set to be covered is the total information contained across all of the spectral bands available. The subsets that can be chosen from are each of the color bands B2-B7, and two different metrics were devised to estimate the information contained within each band.

The first metric used was the mean classification accuracy from the single band testing. Colors with the highest single band classification accuracy were added until no colors remained. Another more refined metric quantified the similarity of the images in each band's dataset. This correlation metric was calculated by comparing the pixels of each tile in the validation set against all other colors. The metric also took into account the single band classification accuracy to ensure that while the colors are dissimilar, they each carry useful information. The end result is the selection of colors which are the most dissimilar to the already selected colors.

$$\rho X, Y = \frac{cov(X, Y)}{\sigma_X \sigma_Y} \tag{6}$$

$$cov: Covariance, \sigma_X, \sigma_Y: Standard\ Deviation\ X, Y$$

$$Accuracy\ Metric: \max(CA_i)\ for\ all\ Bands \tag{7}$$

$$CA_i = classification\ accuracy\ Band_i$$

$$Correlation\ Metric: \min\left(\frac{\rho_{ij}}{CA_i}\right) for\ all\ Bands \tag{8}$$

$$\rho_{ij} = correlation\ between\ Band_i \& Band_j$$

Once again multiple runs were done for each type of network created and for each of the greedy algorithm metrics. From these runs the average classification accuracy was calculated for all networks.

IV. Analysis and Results

Experiment One

The results from the 1st experiment are shown in Figure 19 and Table 8 below. The training algorithm utilized a Stochastic Gradient Descent (SGD) optimizer [36] with a learning rate of 0.0001 and a momentum value of 0.9. Networks were generated for each color and trained for a full 60 epochs before stopping. Test metrics were then collected using the models generated after the 60 epoch cutoff.

Table 7. Experiment #1 Hyper-Parameters and Test Configuration

Experiment #1 Hyper-Parameters	
Architecture	VGG-19
Optimizer	SGD with momentum of .9
Learning Rate	0.0001
Epochs	60
Batch Size	64
Dataset Size	12,084
Training Set	70%
Test Set	30%
Pre-Training	ImageNet
Early Stopping	None

During the training phase all networks were able to reach 100% training accuracy and validation accuracy peaked at around 95% within 40 epochs for most bands. Classification accuracy on the test set was fairly uniform with Band 6 – SWIR #1 seeing the lowest accuracy of all bands. Band 6 also saw greater instability during the training phase, with its accuracy varying widely. The composite RGB band saw no benefit over the grayscale networks even though it contained 3x the data of the grayscale networks. One observed benefit of the additional colors was a more stable accuracy during training.

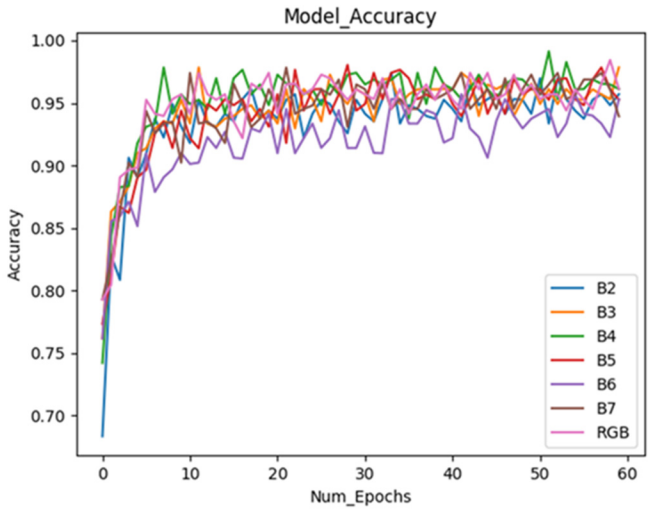

Figure 19. Test accuracy over time for 1st experiment

Table 8. Test classification accuracy at epoch 60

Experiment #1 Classification Accuracy	
Band	**Accuracy**
B2 - Blue	95.8%
B3 - Green	95.8%
B4 - Red	96.8%
B5 - NIR	96.3%
B6 – SWIR #1	93.5%
B7 – SWIR #2	96.1%
Composite RGB	96.1%

The uniform performance of the networks across bands and the observed ineffectiveness of color in increasing accuracy were unexpected results. Overall the classification accuracy was very good and all networks were able to distinguish the large airports from scenery over 93% of the time. The equality in performance could be explained by several factors. The first is that all networks were pre-trained with the ImageNet dataset, which contains over a million images and this coupled with the large number of parameters in VGG-19 leaves little chance for this dataset to have an effect. In fact out of the box the ImageNet features worked very well at identifying the new large airport class. Furthermore since the backpropagation was locked to only the fully connected layers, variation between networks was controlled even further. Another factor could be the dataset itself which contained only the largest airports and randomly sampled background tiles. Since these airports are bigger, it is expected that they will have more distinctive features for the network to extract. The background or scenery tiles also contain statistically “easier tiles”, which is due to the fact that most of the earth is ocean or wilderness. When encountered with undeveloped areas a classification is likely very easy as open plains is very different that paved airport runways. If the scenery tiles were selected to only represent urban areas instead of a random sample then the

classification accuracy could fall. The ambivalence of the networks to color can also be attributed to these factors and as will be seen in experiment two, when medium airports are included color becomes more noticeably important.

Experiment Two

The second experiment used a different optimizer than the first because training accuracy could not progress past random odds with an SGD optimizer. Instead an RMSProp with a learning rate of 0.0001 was used. The experiment started by training grayscale networks for Bands 2 through 8 several times. Training was run for a full 200 epochs with the best weights being saved. Saved models were determined based on their validation accuracy at the end of each epoch. If a model outperformed a previous version it was saved, if not it was passed over. Training error and validation error were logged at each epoch and saved for later analysis. Following the conclusion of each training run the test set was used to collect classification accuracy against the two classes. Mean accuracy values were computed once all training runs were completed.

Table 9. Experiment #2 Hyper-Parameters and Test Configuration

Experiment #2 Hyper-Parameters	
Architecture	SqueezeNet
Optimizer	RMSProp with decay of $1e^{-6}$
Learning Rate	0.0001
Epochs	200
Batch Size	64
Dataset Size	132,426
Training Set	70%
Validation Set	10%
Test Set	20%
Pre-Training	None
Early Stopping	Peak Validation Accuracy

Figure 20 shows the training and validation accuracy for four example runs. The blue lines are the training accuracy and the orange lines the validation accuracy. The networks selected are one visible band, one infrared band, one double resolution band, and one multi-color network. These training curves demonstrate the typical divergence between validation and training accuracy, which indicate that some overfitting is occurring. The multi-color networks tend to have more stability in their validation accuracy in comparison to the grayscale networks. Resolution however does not provide the same type of stability and exhibits wide swings in accuracy.

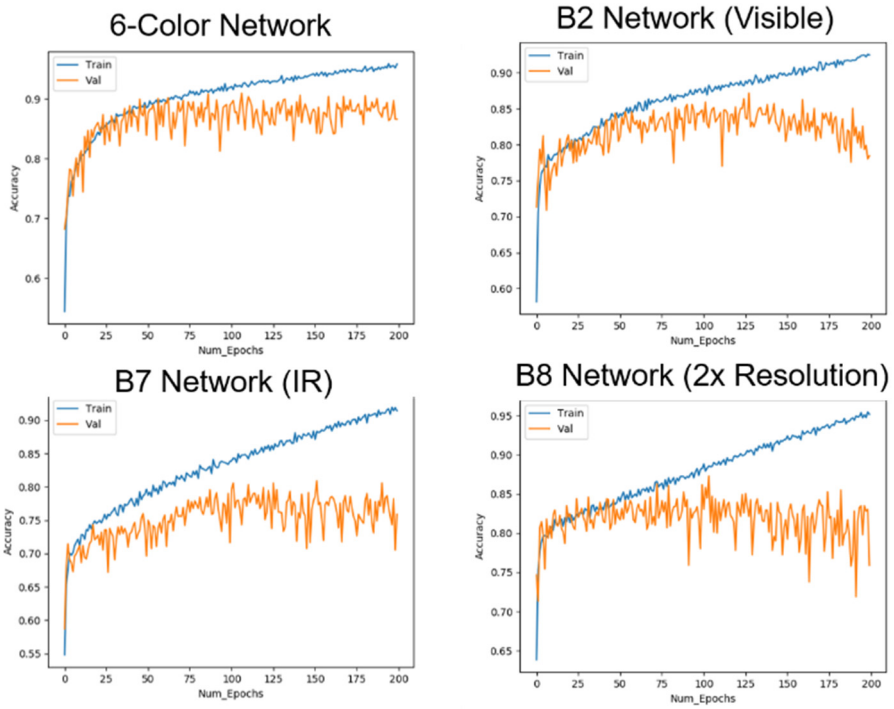

Figure 20. Training and validation accuracy over time for selected networks

Results from experiment #2 are shown in Figures 21-22 and Tables 10-12. Figure 21 shows the validation accuracy during training for all networks. In total this was 7 grayscale networks and 5 multi-color networks. The multi-color networks in Figure 21 were created using the greedy classification accuracy method. One interesting feature of

the validation curves and the test results is that the networks generally fall into three accuracy categories: infrared, visible, and multi-color. Infrared networks tended to have the lowest accuracy as shown by the pink line in Figure 21 and the blue line in figure 22. The infrared networks were comprised of Bands 5-7 and had mean accuracies of 75.39%, 74.86%, and 76.50% respectively. Of all networks trained the SWIR #1 band performed the worst on average. The second grouping was the visible networks, consisting of Bands 2-4 which had slightly higher mean accuracy numbers of 83.83%, 81.78%, and 79.31%. Once again the validation accuracy and test accuracy were highly correlated. Band 8, the panchromatic band performed about as well as the visible bands, but the higher resolution did not seem to offer any benefit. The mean accuracy for this band was only 81.92% and was easily surpassed by even a 2-color network at half the resolution. The last and best performing grouping of networks was the multi-color models. Accuracy for these models depended on the greedy metric chosen, and the results for each metric are shown in Tables 11 and 12.

For the greedy classification accuracy method the two color network showed a small improvement compared to the most accurate grayscale network. The accuracy of the network climbed steadily from 84.84% to a peak of 90.27% with 5-colors. The 6-color network for the greedy classification accuracy approach showed a slight decrease in accuracy from the 5-color network. With the correlation metric a different classification accuracy trend was observed. Accuracy for the 2-color network immediately increased to 89.49% and then plateaued as all the colors were added. This is about where the peak accuracy was recorded using the other color selection method. This immediate jump in performance validates the effectiveness of a correlation metric over raw accuracy when

making the color selections. The conclusion that can be drawn is that the information contained in many of the color bands is redundant, with only a few bands able to provide all of the salient features to the network. For the bands analyzed in this research it was found that the combination of Blue and NIR colors were the most effective for airport classification. This is somewhat surprising as the NIR network by itself had quite poor performance.

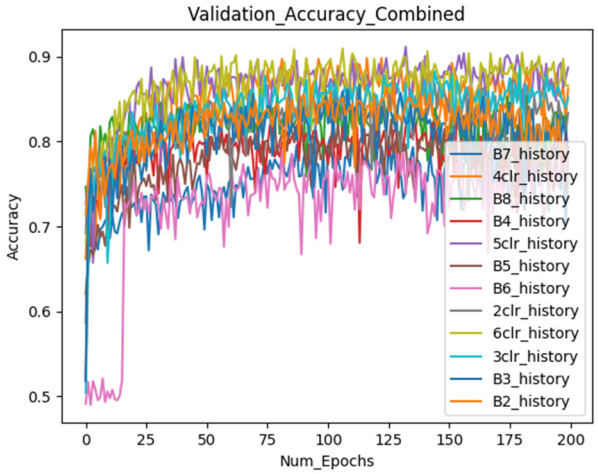

Figure 21. Validation accuracy over time for all models tested during 2nd experiment (Greedy CA method)

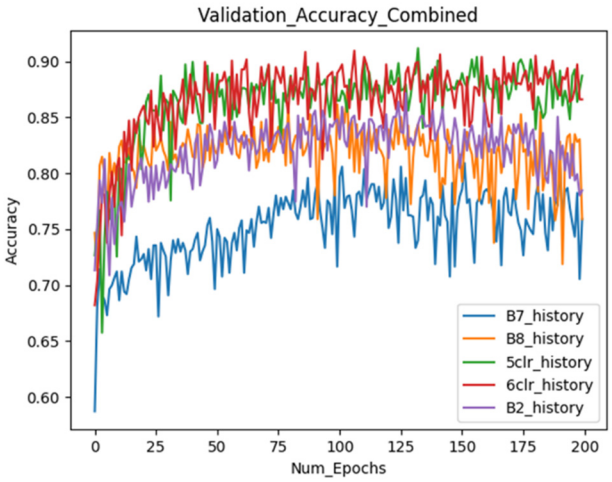

Figure 22. Validation accuracy over time for subset of tested models during 2nd experiment (Greedy CA method)

Table 10. Test classification accuracy for grayscale networks in 2nd experiment

Experiment #2 Grayscale Classification Accuracy	
Band	**Mean Accuracy / Count**
B2 - Blue	83.83% (10)
B3 - Green	81.78% (11)
B4 - Red	79.31% (12)
B5 - NIR	75.39% (8)
B6 – SWIR #1	74.86% (7)
B7 – SWIR #2	76.50% (10)
B8 - Panchromatic	81.92% (12)

Table 11. Test classification accuracy for CA multi-color networks

Experiment #2 Composite Classification Accuracy Greedy Single Band CA	
Selected Bands	**Accuracy / Count**
2-Color B2, B3	84.82% (1)
3-Color B2, B3, B4	87.26% (1)
4-Color B2, B3, B4, B7	87.94% (1)
5-Color B2, B3, B4, B5, B7	90.27% (1)
6-Color B2, B3, B4, B5, B6, B7	88.89% (1)

Table 12. Test classification accuracy for composite color networks

Experiment #2 Composite Classification Accuracy Greedy Correlation Metric	
Selected Bands	**Mean Accuracy / Count**
2-Color B2, B5	89.49% (7)
3-Color B2, B5, B7	89.14% (7)
4-Color B2, B5, B7, B3	89.77% (5)
5-Color B2, B5, B7, B3, B6	90.17% (6)
6-Color B2, B5, B7, B3, B6, B4	89.12% (6)

Also Band 4 straddles the IR and visible groupings, while the IR and visible bands themselves are mutually exclusive. Figure 24 visually shows how the probability distribution of the multi-color networks are highly correlated. No statistical distinction in classification accuracy can be drawn from the multi-color networks for the greedy correlation color selection method.

Table 13. Comparison of the 2-Color greedy correlation metric network to other networks

Mean Classification Accuracy 2-Color Network (Correlation Metric) Student T-Test: Two-tailed, Two Sample Unequal Variance	
Networks	**P-Value**
2-Color & B2	0.0000
2-Color & B3	0.0000
2-Color & B4	0.0000
2-Color & B5	0.0000
2-Color & B6	0.0000
2-Color & B7	0.0000
2-Color & B8	0.0000
2-Color & 3-Color	0.2786
2-Color & 4-Color	0.5516
2-Color & 5-Color	0.2912
2-Color & 6-Color	0.6038

Table 14. Comparison of the B8 network to other tested networks

Mean Classification Accuracy B8 Network (Correlation Metric) Student T-Test: Two-tailed, Two Sample Unequal Variance	
Networks	**P-Value**
B8 & B2	0.0001
B8 & B3	0.7184
B8 & B4	0.0000
B8 & B5	0.0000
B8 & B6	0.0000
B8 & B7	0.0000
B8 & B8	0.0000
B8 & 3-Color	0.0000
B8 & 4-Color	0.0000
B8 & 5-Color	0.0000
B8 & 6-Color	0.0000

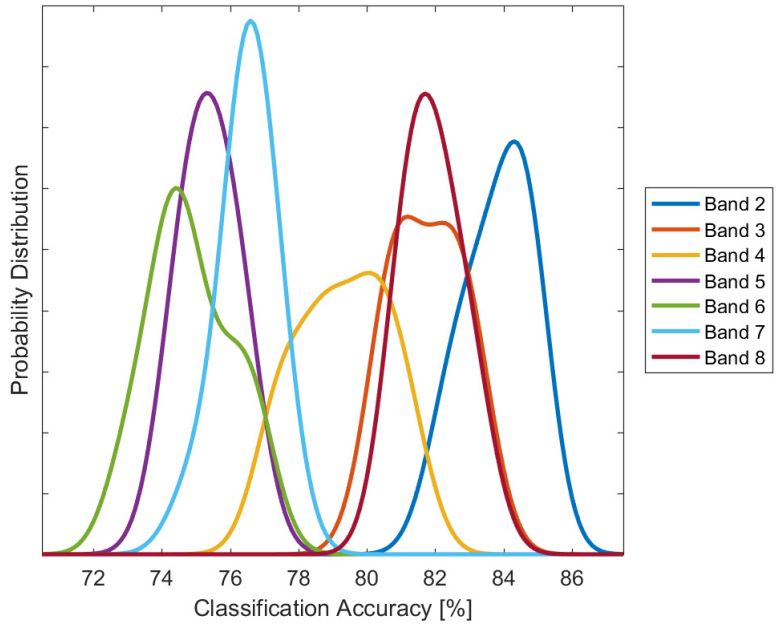

Figure 23. Probability distribution of test classification accuracy for grayscale networks

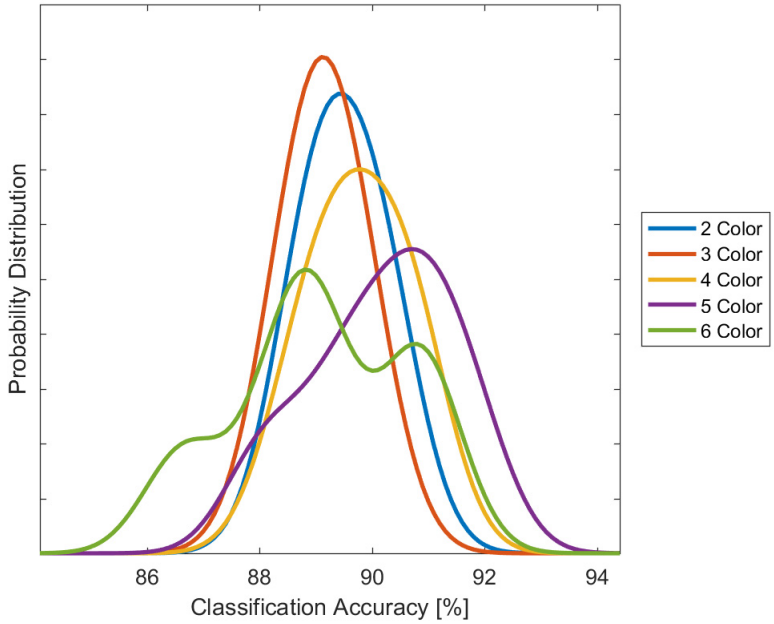

Figure 24. Probability distribution of test classification accuracy for multi-color networks (Greedy Correlation Method)

Finally Figure 25 compares the two color selection metrics against each other. The blue line shows the mean test classification accuracy for the greedy correlation method, and the orange line shows the test classification accuracy for the grayscale classification accuracy method. This chart highlights the advantage of the correlation metric, which is able to reach maximum accuracy with 2-colors, while the grayscale classification accuracy metric requires 5-colors.

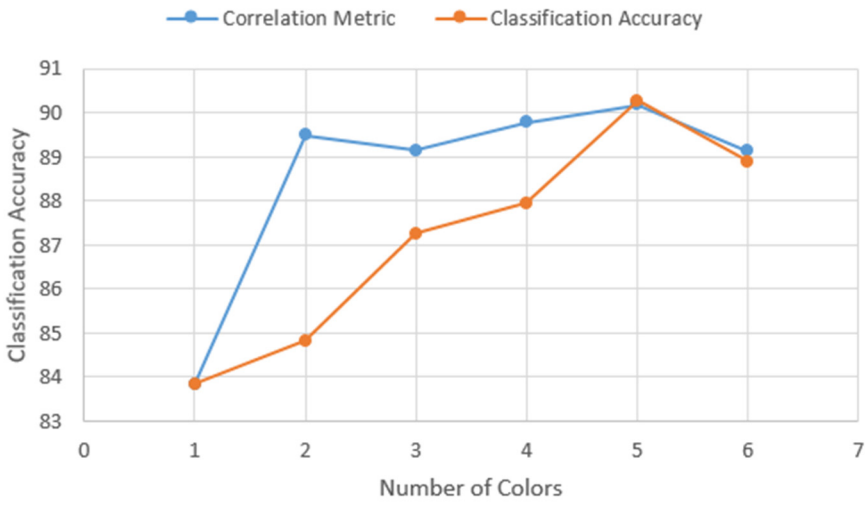

Figure 25. Comparison of greedy algorithm metrics

Overall the classification accuracy in the second experiment was much lower than the first experiment. The highest recorded accuracy for experiment #1 was 96.8% accuracy compared the second experiments 90.27%. Even the lowest accuracy from the first experiment exceeded the best second experiment results.

This delta could be due to several factors, with the first being that the dataset itself was more difficult in experiment two than in experiment one. The second experiment included more airports and more varied airports. The inclusion of medium size airports meant that much smaller and less distinct airports needed to be characterized. A second

source of lower accuracy could be the combination of the SqueezeNet architecture and the inability to include ImageNet pre-training in experiment two. The net effect of these two choices is a dataset with less total examples and a network with less capacity. SqueezeNet itself is benchmarked to perform as well as AlexNet against the ImageNet dataset, while the VGG-19 architecture exceeds AlexNet performance by about 7%. A margin of 6.96% was observed between the 3-color network in experiment which used VGG-19 and the greedy correlation metric 3-color network using SqueezeNet. Taking into account the more than 50x reduction in parameter count between SqueezeNet and VGG-19, the efficiency of the network in experiment two is exceptionally good.

V. Conclusion

Future Work

Several open avenues remain for further research in RS target detection. The logical next step for this research is to move from a classification to an image segmentation or image localization approach. The completed experiments lay a solid groundwork for pursuing this option, with the classifiers having the ability to act as a component of a larger localization system. Some preliminary work has already been done to test the feasibility for the models trained in experiment two to detect targets within full size Landsat-8 images. This consisted of taking a trained model and feeding it tiles from a larger image via a sliding window. More work needs to be done to better tailor the contents of the dataset to improve localization accuracy and not classification accuracy. One recommendation is to utilize the adjacent tiling algorithm in creating a dataset instead of the all-tiling algorithm and to avoid sampling from the scenery tiles. A

weighted backpropagation approach based on the relative class size should be explored to compensate for class imbalance. Furthermore in future experiments whose goal is maximum accuracy a network architecture other than SqueezeNet should be used, two promising architectures for high accuracy would be the ResNet and GoogleNet architectures. Finally another technique that would likely yield better model results is synthetic data augmentation, which increases the size of the overall dataset via plausible image transformations such as rotations and offsetting.

Localization itself also opens up new possibilities in how CNNs can present their results to human analysts. One particularly interesting method is the creation of detection heatmaps which can be overlaid on the source imagery to aid the location of unknown targets. By using trained models probability estimates can be created by cutting a larger image up into tiles via the sliding window technique and recording the raw results from the output layer. In this way a heatmap for each class can be created and overlaid on the source imagery, highlighting where the neural network believes targets exist.

Another avenue for exploration is the classification and localization of non-stationary targets such as ships, planes, and cars. These target types would be more challenging than a large stationary airports, requiring more sophisticated automated dataset generation techniques. Due to the targets not being fixed over time locating truth imagery would become more difficult. The spatial location would still need to be correlated with available imagery as before via coordinates and a new temporal location would also need to be correlated with imagery. These new targets types would also require different higher resolution imagery sources; two potential commercial options that exist are the DigitalGlobe and Planet Labs constellations.

Recommendations

The research results show that both low parameter embedded style architectures and complex deep architectures can successfully classify airports in satellite imagery with a 30 meter spatial resolution. The VGG-19 network in experiment one saw very high test classification accuracy and short training times in part due to the ImageNet pre-training. This implies that that feature extractors which excel in the ImageNet challenge also excel at extracting features from RS imagery. Another important outcome was that for RS imagery, hand-labeling of pictures can be avoided completely if a source of target coordinates can be found. All networks during the experimentation phase were trained using automatically cropped truth data which allowed test classification accuracy to reach over 90% with both the VGG-19 and SqueezeNet architectures. Coordinate based automated dataset generation addresses one of the barriers in applying CNNs to RS imagery by allowing the acquisition of large datasets. One final observation is that when it comes to increasing classification accuracy simply obtaining higher resolution images does not necessarily lead to better results. This research shows that for the detection of airports in Landsat-8 imagery a network using double the resolution will be outperformed by a network using two colors at a half the resolution. Furthermore, simply adding more colors to the imagery also does not guarantee the best performance increase. Instead picking colors which contrast and provide meaningful salient information will perform as well as many colors. These last two points are informative to sensor designers who are developing future sensors with AI classification in mind.

Appendix

The following files are code excerpts for the Automated Dataset Generation.

dataset_gen_master.py

```
# This is the top level script for creating a new dataset from raw
Landsat imagery

import os
from fix_airport_names import *
from AutoTiling_Multicore import *
from autotiling_check import *
from darkness_filter import *
from consolidate_bands import *
from gen_npy_images import *
from dictionary_gen import *

# ======================
# Configuration Variables
raw_data_path = '/datatank/raw_landsat_data' # Path to raw imagery
location for CSV target checking
csv_path = '/media/afit/Seagate Backup Plus Drive/Landsat Download
Lists' # Path to clean CSV target lists
load_path = '/home/afit/Desktop/Batches' # Path to load raw images from
for tiling
tile_path = '/home/afit/Desktop/Tiles' # Base path to save generated
tiles to
band_path = '/home/afit/Desktop/Bands' # Base path to consolidate each
band's tiles to
dataset_path = '/home/afit/Desktop/Datasets/6clr' # Base path to the
final dataset location

tile_size = 256 # Size in pixels of generated tiles
train_pct = .7 # Percentage of images to be used for training (Sum to
1.0)
val_pct = .1 # Percentage of images to be used for validation (Sum to
1.0)
test_pct = .2 # Percentage of images to be used for testing (Sum to
1.0)
channel_colors = ['B2', 'B5', 'B7', 'B3', 'B6', 'B4'] # List of what
colors to make an image with (B2-B8)

tile_all = False # Variable to indicate whether non_target tiles should
be generated
gen_DG_cords = False # Variable to indicate if a coordinate CSV should
be generated for target tiles
test_targets_only = True # Variable that controls whether the black &
grey test is applied to non target tiles
gen_missing_bands = False # Variable that indicates if you want a
listing of missing bands for re-downloading
pickle_dataset = False # Set to true if you want to serialize the
dataset
```

```
# ======================

# ======================
# Debug Variables
step_one = False
step_two = False
step_three = False
step_four = False
step_five = True
step_six = True
# ======================

# STEP #1: Run error checking & correction scripts against the target
lists
if step_one:
    # Fix existing download lists
    existingCSVFix(raw_data_path)
    # Fix unused download lists
    CSVFix(csv_path)

# STEP #2: Run auto-tiling script & generate target/non_target tiles
if step_two:
    batches = os.listdir(load_path)
    for batch in batches:
        # Create load path, target list, and save path based on batch
        load_path_batch = os.path.join(load_path, batch)
        csv_name = 'airports_subset_' + batch.split(' ')[1] + '.csv'
        target_list = os.path.join(load_path_batch, csv_name)
        save_path_batch = os.path.join(tile_path, batch + ' tiles')
        # Run tiling code
        autoTile(target_list, load_path_batch, save_path_batch,
tile_size=tile_size, tile_all=tile_all,
                 gen_DG_cords=gen_DG_cords)
        # Check that the number of target tiles generated matches the
expected amount from CSV
        checkTargets(target_list, save_path_batch)

# STEP #3: Remove tiles that are outside of the actual images bounds or
corrupted (Target tiles only)
if step_three:
    batches = os.listdir(tile_path)
    for batch in batches:
        # Run black & grey filter on each batch of tiles
        total_images, total_num_black, total_num_grey =
removeBadTiles(os.path.join(tile_path, batch),

targetOnly=test_targets_only)
        print('===============', batch, '===============')
        print('Number of starting images:', total_images)
        print('Number of removed black images:', total_num_black)
        print('Number of removed grey images:', total_num_grey)

# STEP #4: Consolidate batch tiles into shared folders, filter out
partial band pairs, and sync num tiles per class
if step_four:
```

```
    num_files_moved = consolidate(tile_path, band_path,
genMissing=gen_missing_bands)
    print(num_files_moved, 'from all classes moved')

# STEP #5: Generate Images as numpy arrays using selected colors, split
into train/validation/test sets
if step_five:
    createDataset(band_path, dataset_path, channel_colors,
img_size=tile_size,
                  train=train_pct, val=val_pct, test=test_pct)

# STEP #6: Create dictionary files with path & class information
if step_six:
    createDict(dataset_path)
```

fix_airport_names.py

```
# This script modifies target_csv airport names to replace '/'
characters with '-' characters
# It also checks for and removes duplicate target entries
# This change avoids filename errors when executing the auto-tiling
script

import os
import pandas as pd

# Remove duplicate entries & rebuild dataframe cells
def removeDuplicate(string, index):
    temp = []
    split_str = string.split('|')
    split_str[index] = None
    for item in split_str:
        if item != None:
            temp.append(item)
            temp.append('|')
    temp = temp[:-1]
    return ''.join(temp)

def existingCSVFix(basePath):

    num_csv_files = 0
    num_mod_targets = 0
    num_removed = 0

    batch_folders = os.listdir(basePath)

    print("==================================================")
    print("Checking existing download lists for target errors")
    print("==================================================")

    # Find each csv file in existing downloads
    for batch_fldr in batch_folders:
        dir_contents = os.listdir(os.path.join(basePath, batch_fldr))
        for item in dir_contents:
            filename = os.path.join(basePath, batch_fldr, item)
```

```
            if os.path.isfile(filename):
                # Open csv file & fix target names
                num_csv_files += 1
                # print("==============================")
                # print("Fixing", item)
                # print("==============================")
                csv = pd.read_csv(filename)
                # Split each row into individual targets
                for i, target in enumerate(csv['name']):
                    split_target_names = str(target).split('|')
                    new_names = []
                    unique_names = set()
                    num_unique_names = 0
                    repeat_rmv = 0
                    # Check each target name for invalid character
('/')
                    for x, target_name in
enumerate(split_target_names):
                        new_name = target_name.replace('/', '-')
                        # See if this name was modified
                        if new_name != target_name:
                            num_mod_targets += 1
                            # print(target_name, "===modified to==>",
new_name)
                        # Check that all names are unique
                        unique_names.add(new_name)
                        num_unique_names += 1
                        if len(unique_names) != (num_unique_names -
repeat_rmv):
                            # Have a repeat name & need to remove entry
                            csv_row = csv.iloc[i]
                            # print("Duplicate target", new_name,
"removed at row", i + 2)
                            # Remove duplicate entries
                            type = removeDuplicate(csv_row['type'], x -
repeat_rmv)
                            lat = removeDuplicate(csv_row['lat'], x -
repeat_rmv)
                            lon = removeDuplicate(csv_row['lon'], x -
repeat_rmv)
                            alt = removeDuplicate(csv_row['alt'], x -
repeat_rmv)
                            # Reassign fixed row
                            csv.set_value(i, 'type', type)
                            csv.set_value(i, 'lat', lat)
                            csv.set_value(i, 'lon', lon)
                            csv.set_value(i, 'alt', alt)
                            # Update removal counter
                            repeat_rmv += 1
                        else:
                            # Name is not repeat add to list for
merging at loop exit
                            new_names.append(new_name)
                            new_names.append('|')
                    # Update total counter of duplicate targets removed
```

```
                    num_removed += repeat_rmv
                    # Rebuild CSV row
                    # Remove last '|' character
                    new_names = new_names[:-1]
                    # Find row index
                    row_index = csv.loc[csv.name == target].index[0]
                    # Write new value to dataframe
                    csv.set_value(row_index, 'name',
''.join(new_names))
                # Write all updates to file
                csv.to_csv(filename, index=False)

    print("\n====Program Results=====")
    print("Read", num_csv_files, "downloaded csv files, fixed",
num_mod_targets, "target names, and removed",
          num_removed, "duplicate targets\n")

def CSVFix(basePath):
    num_mod_targets = 0
    num_removed = 0

    files = os.listdir(basePath)
    num_csv_files = len(files)

    print("========================================")
    print("Checking download lists for target errors")
    print("========================================")

    # Loop through each CSV file
    for file in files:
        # print("==============================")
        # print("Fixing", file)
        # print("==============================")
        filename = os.path.join(basePath, file)
        csv = pd.read_csv(filename)
        # Split each row into individual targets
        for i, target in enumerate(csv['Name']):
            split_target_names = str(target).split('|')
            new_names = []
            unique_names = set()
            num_unique_names = 0
            repeat_rmv = 0
            # Check each target name for invalid character ('/')
            for x, target_name in enumerate(split_target_names):
                new_name = target_name.replace('/', '-')
                # See if this name was modified
                if new_name != target_name:
                    num_mod_targets += 1
                    # print(target_name, "===modified to==>", new_name)
                # Check that all names are unique
                unique_names.add(new_name)
                num_unique_names += 1
                if len(unique_names) != (num_unique_names -
repeat_rmv):
                    # Have a repeat name & need to remove entry
```

```
                    csv_row = csv.iloc[i]
                    # print("Duplicate target", new_name, "removed at
row", i + 2)
                    # Remove duplicate entries
                    type = removeDuplicate(csv_row[' Type'], x -
repeat_rmv)
                    lat = removeDuplicate(csv_row[' Lat'], x -
repeat_rmv)
                    lon = removeDuplicate(csv_row[' Lon'], x -
repeat_rmv)
                    alt = removeDuplicate(csv_row[' Alt'], x -
repeat_rmv)
                    # Reassign fixed row
                    csv.set_value(i, ' Type', type)
                    csv.set_value(i, ' Lat', lat)
                    csv.set_value(i, ' Lon', lon)
                    csv.set_value(i, ' Alt', alt)
                    # Update removal counter
                    repeat_rmv += 1
                else:
                    # Name is not repeat add to list for merging at
loop exit
                    new_names.append(new_name)
                    new_names.append('|')
            # Update total counter of duplicate targets removed
            num_removed += repeat_rmv
            # Rebuild CSV row
            # Remove last '|' character
            new_names = new_names[:-1]
            # Find row index
            row_index = csv.loc[csv.Name == target].index[0]
            # Write new value to dataframe
            csv.set_value(row_index, 'Name', ''.join(new_names))
        # Write all updates to file
        csv.to_csv(filename, index=False)

    print("\n====Program Results=====")
    print("Read", num_csv_files, "csv files, fixed", num_mod_targets,
"target names, and removed",
          num_removed, "duplicate targets\n")
```

Autotiling_Multicore.py

```
import os
import pandas as pd
from datetime import datetime
from multiprocessing import Pool
import AutoTiling_Functions
import DigiGlobe_Coord

def autoTile(csv_path, Directory, storage_location, tile_size=256,
target_tiles='airports', non_target_tiles='non_airports',
             tile_all=True, gen_DG_cords=True):

    # Create paths for folders if they do not exist
```

```
    if not os.path.exists(storage_location):
        os.makedirs(storage_location)
    target_path = storage_location + '/' + target_tiles
    if not os.path.exists(target_path):
        os.makedirs(target_path)
    all_targets = target_path + '/all_' + target_tiles
    if not os.path.exists(all_targets):
        os.makedirs(all_targets)
    non_target_path = storage_location + '/' + non_target_tiles
    if not os.path.exists(non_target_path):
        os.makedirs(non_target_path)

    ##########################################
    # Start Timing of Advanced Cropping Algorithm
    startTime = datetime.now()

    # Read target list into dataframe
    target_info = pd.read_csv(csv_path)

    # Find number of physical cores to determine number of worker
processes to create
    cpu_info = dict()

    # Finds the number of physical cores on a linux computer
    # Source:
https://github.com/teamdiamond/analysis/blob/master/lib/qutip/hardware_
info.py
    for l in [l.split(':') for l in os.popen('lscpu').readlines()]:
        cpu_info[l[0]] = l[1].strip('.\n ').strip('kB')
    sockets = int(cpu_info['Socket(s)'])
    cores_per_socket = int(cpu_info['Core(s) per socket'])
    num_processes = sockets * cores_per_socket * 5 // 4  # Only do this
with hyperthreading, otherwise remove the '* 5//4'
    print("Creating", num_processes, "worker processes")

    # Split the csv rows as evenly as possible into X processes
    row_count = len(target_info.index)
    floor_inc = row_count // num_processes
    ceil_inc = (row_count + num_processes) // num_processes
    num_ceil = int((row_count - (floor_inc*num_processes)) / (ceil_inc
- floor_inc))
    num_floor = int(num_processes - num_ceil)

    # --Debug Test--
    # print(floor_inc, ceil_inc)
    # print(num_floor, num_ceil)

    # Create a list of inputs for each worker process
    input_list = []
    current_row = 0
    for floor in range(0, num_floor):
        input_list.append([current_row, current_row + floor_inc - 1,
target_info, Directory, tile_size, tile_all, target_path,
non_target_path, all_targets])
        current_row += floor_inc
```

```
    for ceil in range(0, num_ceil):
        input_list.append([current_row, current_row + ceil_inc - 1,
target_info, Directory, tile_size, tile_all, target_path,
non_target_path, all_targets])
        current_row += ceil_inc

    # Create a number of worker processes 5/4 times the number of cores
and run the tiling function over all of them
    #if __name__ == '__main__':
    with Pool(processes=num_processes) as p:
        p.starmap(AutoTiling_Functions.Tile_Targets, input_list)

    # Calculate and print time taken to tile images
    time_taken = datetime.now() - startTime
    print('\nTime taken to tile Landsat images: ' + str(time_taken))

    # Generate DigiGlobe target coordinate csv
    if gen_DG_cords == True:
        DigiGlobe_Coord.generate_coord_csv(all_targets + '/')
```

Autotiling_Functions.py

```
import os
import numpy as np
from osgeo import gdal
from osgeo import osr
from PIL import Image


#########################################################################
################################################
# http://monkut.webfactional.com/blog/archive/2012/5/2/understanding-
# raster-basic-gis-concepts-and-the-python-gdal-library/
#########################################################################
################################################

# Returns coordinates converted into the format supported by gdal
libraries
def transform_wgs84_to_utm(lon, lat):
    def get_utm_zone(longitude):
        return int(1 + (longitude + 180.0) / 6.0)

    def is_northern(latitude):
        """
        Determines if given latitude is a northern for UTM
        """
        if latitude < 0.0:
            return 0
        else:
            return 1

    utm_coordinate_system = osr.SpatialReference()
    utm_coordinate_system.SetWellKnownGeogCS("WGS84")  # Set geographic
coordinate system to handle lat/lon
    utm_coordinate_system.SetUTM(get_utm_zone(lon), is_northern(lat))
```

```
    wgs84_coordinate_system = utm_coordinate_system.CloneGeogCS()  #
Clone ONLY the geographic coordinate system

    # Create transform component
    wgs84_to_utm_geo_transform =
osr.CoordinateTransformation(wgs84_coordinate_system,
utm_coordinate_system)  # (, )
    return wgs84_to_utm_geo_transform.TransformPoint(lon, lat, 0)  #
Returns easting, northing, altitude


# Finds the number of targets in the string grouping
def find_num_targets(names):
    result = 1
    for char in range(len(names)):
        if names[char] == '|':
            result += 1
    return result


# Finds the specific target value based on separator_num
def find_target_value(value, separator_num, max_separators):

    # Initialize variables to track the position of the desired target
value
    beginning_sep, ending_sep = 0, 0
    counter = 0
    endpoint = len(value)

    # Find the correct value in the string separated by '|'
    for position in range(endpoint):
        if value[position] == '|':
            if counter <= separator_num:
                beginning_sep = ending_sep
                ending_sep = position
                counter += 1
    if separator_num == 0 and max_separators == 0:
        result = value
    elif separator_num == 0:
        result = value[beginning_sep:ending_sep]
    elif separator_num == max_separators:
        result = value[ending_sep+1:]
    else:
        result = value[beginning_sep+1:ending_sep]
    return result


# Returns a set of tiles that should not be cut from the landsat image
def tile_non_targets(entityID, band, tile_save_path, target_coord,
size, directory):

    # Create the Landsat path and the specific storage directory for
this entityId
    landsat_image_path = directory + '/' + entityID + '/' + band
```

```
    if not os.path.exists(tile_save_path + '/' + entityID + '/'):
        os.makedirs(tile_save_path + '/' + entityID + '/')

    # Open Geotiff
    landsat_geotiff = gdal.Open(landsat_image_path)

    # Grab corner coordinates & resolution from metadata
    # xMin and yMax make up the origin, xRes and yRes are the pixel
size
    # xMin, xRes, xSkew, yMax, ySkew, yRes = geotiff.GetGeoTransform()
    xMin_func, xRes_func, xSkew_func, yMax_func, ySkew_func, yRes_func
= landsat_geotiff.GetGeoTransform()
    xMax_func = xMin_func + (landsat_geotiff.RasterXSize * xRes_func)
    yMin_func = yMax_func + (landsat_geotiff.RasterYSize * yRes_func)
    x_resolution = xRes_func

    # Determine the number of tiles on x and y axis of landsat image
    num_x = np.int((xMax_func - xMin_func) / (size * x_resolution))
    num_y = np.int((yMax_func - yMin_func) / (size * x_resolution))

    # find a value that will be used to check if the tile will be saved
or not
    checker = round((num_x * num_y // 100) ** .5)

    # Loop to tile a Landsat-8 Image Preserving Metadata
    tileyMax_func = yMax_func
    for y_position in range(num_y):
        tileyMin_func = tileyMax_func - (size * x_resolution)
        tilexMin_func = xMin_func
        for x_position in range(num_x):
            tilexMax_func = tilexMin_func + (size * x_resolution)

            # Saves a small porition of the non-target tiles (those
that pass the mod check with checker) and checks to make sure they do
not overlap with targets, does not save the tile if they do.
            if (x_position % checker == 0) and (y_position % checker ==
0):
                for k in range(0, len(target_coord)):

                    # Check that the tile's coordinates do not overlap
with any of the target's coordinates, pass if it does
                    if (target_coord[k][0] < tilexMin_func <
target_coord[k][2] or target_coord[k][0] < tilexMax_func <
                            target_coord[k][2]) and (
                                        target_coord[k][1] <
tileyMin_func < target_coord[k][3] or target_coord[k][
                                    1] < tileyMax_func <
target_coord[k][3]):
                        pass
                    else:

                        # Save the tile as a geotiff file
                        image_options = gdal.WarpOptions(options=[],
format='GTiff',
```

```
outputBounds=[tilexMin_func, tileyMin_func, tilexMax_func,

tileyMax_func])
                        tile_filename = tile_save_path + '/' + entityID
+ '/' + 'Tile: ' + str(x_position) + '.' + str(y_position) +
band[len(entityID):]
                        gdal.Warp(tile_filename, landsat_image_path,
options=image_options)

                        # Check that the tile has no black pixels, and
if it does delete the image and add it to the cut_check set
                        img = Image.open(tile_filename)
                        pixels = img.load()  # get the pixels as a
flattened sequence
                        black_thresh = 50
                        has_black = False
                        if pixels[0, size - 1] < black_thresh or
pixels[0, 0] < black_thresh or pixels[size - 1, 0] < black_thresh or
pixels[size - 1, size - 1] < black_thresh:
                            has_black = True

                        if has_black:
                            os.remove(tile_filename)
            tilexMin_func = tilexMax_func
        tileyMax_func = tileyMin_func
    return


# Main function that tiles Landsat images
def Tile_Targets(start_row, end_row, target_info, Directory,
pixel_size, tile_all, target_path, non_target_path, all_targets):
    for targets in range(start_row, end_row + 1):

        # Initialize/Reset the list of target coordinates for the
specific Landsat Image
        coord = []

        # Initialize/Reset a set of targets that are tiled with errors
        empty_targets = set([])

        # Get the grouped values for the targets in this scene
        target_entityId = target_info['entityID'].iloc[targets]
        target_lat_group = str(target_info['lat'].iloc[targets])
        target_lon_group = str(target_info['lon'].iloc[targets])
        target_name_group = target_info['name'].iloc[targets]
        target_type_group = target_info['type'].iloc[targets]

        # Find number of unique targets
        num_targets = find_num_targets(target_name_group)

        # Initialize first_band to True to indicate the process is
working with the first (of possibly many) bands
        first_band = True
```

```
        for file in os.listdir(Directory + '/' + target_entityId):
            if file.endswith(".TIF"):

                # Create path for geotiff file
                image_path = Directory + '/' + target_entityId + '/' +
file

                # Open Geotiff file
                geotiff = gdal.Open(image_path)

                # Grab corner coordinates & resolution from metadata
                # xMin and yMax make up the origin, xRes and yRes are
the pixel size
                # xMin, xRes, xSkew, yMax, ySkew, yRes =
geotiff.GetGeoTransform()
                geoTransform = geotiff.GetGeoTransform()
                px_resolution = geoTransform[1]

                # Convert target lat/lon into generic values (match
landsat bounds for cropping)
                offset = (pixel_size / 2) * px_resolution

                # Creates the target tiles and adds their coordinates
in an array that is cross reference when cutting tiles
                for sep in range(0, num_targets):

                    # Get the lat, lon, name, and type for the specific
target being tiled
                    target_lat = find_target_value(target_lat_group,
sep, num_targets - 1)
                    target_lat = float(target_lat)
                    target_lon = find_target_value(target_lon_group,
sep, num_targets - 1)
                    target_lon = float(target_lon)
                    target_name = find_target_value(target_name_group,
sep, num_targets - 1)

                    if target_name in empty_targets:
                        pass
                    else:

                        target_type =
find_target_value(target_type_group, sep, num_targets - 1)

                        # Create target save location based on type
                        target_save_path = target_path + '/' +
target_type + '/' + target_entityId + '/'
                        if not os.path.exists(target_save_path):
                            os.makedirs(target_save_path)

                        # Convert from WGS84 Lat/Lon --> UTM
                        target_x, target_y, target_z =
transform_wgs84_to_utm(target_lon, target_lat)
```

```
                        # Determine the boundaries of the cropped
target
                        aTilexMin = target_x - offset
                        aTilexMax = target_x + offset
                        aTileyMin = target_y - offset
                        aTileyMax = target_y + offset

                        # Crop and save image, if it is a black box
remove it and add the target name and entityId to an error list
                        options = gdal.WarpOptions(options=[],
format='GTiff',

outputBounds=[aTilexMin, aTileyMin, aTilexMax, aTileyMax])
                        filename = target_save_path + target_name +
file[len(target_entityId):]
                        gdal.Warp(filename, image_path,
options=options)

                        # Save the file to be used to create DigiGlobe
coordinates
                        if first_band:
                            all_filename = all_targets + '/' +
target_name + file[len(target_entityId):]
                            gdal.Warp(all_filename, image_path,
options=options)

                            # Stores coordinates relative to image in
list of tuples
                            coordinates = (aTilexMin, aTileyMin,
aTilexMax, aTileyMax)
                            coord.append(coordinates)

                # Tiles the Landsat Image (without overlapping targets)
if specified to do so in AutoTiling_Multicore
                if tile_all:

                    # Cut out the non_target tiles
                    tile_non_targets(target_entityId, file,
non_target_path, coord, pixel_size, Directory)

                # Set first to false
                first_band = False
```

autotiling_check.py

```
# Script to assess how many targets were expected to be cropped vs how
many were actually generated

import os
import pandas as pd

def checkTargets(csv_filename, basePath):
    # Hardcoded paths for airports only
    airport_path = os.path.join(basePath, 'airports')
    classes = os.listdir(airport_path)
```

```
    classes.remove('all_airports')

    # Start general code
    csv = pd.read_csv(csv_filename)
    num_est_targets = 0
    num_not_tiled_targets = 0

    # Estimated Targets
    for i, target in enumerate(csv['name']):
        # Entity ID
        entity_ID = csv.loc[i, 'entityID']

        if csv.loc[i, 'downloaded'] == 'Y':
            # Number of expected targets
            split_target_names = str(target).split('|')
            est_count = len(split_target_names)
        else:
            est_count = 0

        num_est_targets += est_count

        # Actual number of targets
        for target_type in classes:
            unique_names = set()
            try:
                tiles = os.listdir(os.path.join(airport_path,
target_type, entity_ID))
                for tile in tiles:
                    # See if this tile is a new airport or just another
band
                    unique_names.add(tile.split('_')[0])
                num_unique_names = len(unique_names)
            except:
                num_unique_names = 0
        delta = est_count - num_unique_names
        if delta != 0:
            num_not_tiled_targets += delta
            print(entity_ID, "has", delta, "not tiled")

    # Delta calculation to determine pass/fail
    final_delta = num_not_tiled_targets

    if final_delta == 0:
        print('Passed test')
        return True
    else:
        print('Failed test', final_delta, "targets are missing")
        return False
```

darkness_filter.py

```
# Retrieved from: https://stackoverflow.com/questions/27868250/python-
find-out-how-much-of-an-image-is-black
import os
from PIL import Image
```

```
def removeBadTiles(basePath, targetOnly=True):
    if targetOnly == True:
        subdir_names = ['airports/large_airport',
'airports/medium_airport']
    else:
        subdir_names = ['airports/large_airport',
'airports/medium_airport', 'non_airports']

    total_num_black = 0
    total_num_grey = 0
    total_images = 0

    for path in subdir_names:

        if os.path.exists(os.path.join(basePath, path)):

            subDirs = os.listdir(os.path.join(basePath, path))

            for dir in subDirs:
                files = os.listdir(os.path.join(basePath, path, dir))
                for file in files:
                    im = Image.open(os.path.join(basePath, path, dir,
file))
                    pixels = im.getdata()          # get the pixels as
a flattened sequence
                    black_thresh = 1
                    nblack = 0
                    ngrey = 0
                    for pixel in pixels:

                        if pixel < black_thresh:
                            nblack += 1

                        if pixel > 4990 and pixel < 5010:
                            ngrey += 1
                    n = len(pixels)

                    # Black Pictures (off edge)
                    if (nblack / float(n)) > 0.25:
                        total_num_black += 1
                        print('Image', total_images, 'removed')
                        os.remove(os.path.join(basePath, path, dir,
file))

                    # Gray Pictures
                    elif (ngrey / float(n)) > 0.25:
                        print('Image', total_images, 'removed')
                        total_num_grey += 1
                        os.remove(os.path.join(basePath, path, dir,
file))
                    else:
                        print('Image', total_images, 'passed')

                    total_images += 1
```

```
        else:
            print('Class', path, 'does not exist for this batch')

    return total_images, total_num_black, total_num_grey

def checkTile(path):
    im = Image.open(path)
    pixels = im.getdata()  # get the pixels as a flattened sequence
    black_thresh = 1
    nblack = 0
    ngrey = 0

    for pixel in pixels:

        if pixel < black_thresh:
            nblack += 1

        if pixel > 4990 and pixel < 5010:
            ngrey += 1

    n = len(pixels)

    # Black Pictures (off edge)
    if (nblack / float(n)) > 0.25:
        return False
    # Gray Pictures
    elif (ngrey / float(n)) > 0.25:
        return False
    else:
        return True
```

consolidate_bands.py

```
# Todo: Check that the random selection of non_aiports tiles is working
properly, theres seems to be little variation
# This script moves all tiles from separate entityID folders into
shared folders by class & band

import os
from shutil import copyfile
from darkness_filter import checkTile
from random import randint

# Creates a directory if one doesn't exist
def createDir(path):
    if not os.path.exists(path):
        os.makedirs(path)

# Check if a file exists at the specified path
def fileExist(path):
    if os.path.exists(path):
        return True
    else:
        return False
```

```
def airportsMove(loadPath, savePath):
    # Tracks the number of airports that were moved so that an equal
number of non airports can be moved
    num_airports = 0

    # Make sure every airport has an image in these bands
    band_labels = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8']

    # Airport Sub-Types
    sub_classes = os.listdir(loadPath)
    sub_classes.remove('all_airports')

    for airport_type in sub_classes:
        # Listing of all folders containing images
        entityIDs = os.listdir(os.path.join(loadPath, airport_type))
        for entityID in entityIDs:
            images = os.listdir(os.path.join(loadPath, airport_type,
entityID))
            unique_names = set()
            for img in images:
                # Get names of each airport in directory
                unique_names.add(img.split('_')[0])

            # Check that each airport has a full compliment of bands
(Band 2 - Band 8)
            for name in unique_names:
                move_airport = True
                # Check if each band is present
                for band in band_labels:
                    filename = name + '_' + band + '.TIF'
                    if fileExist(os.path.join(loadPath, airport_type,
entityID, filename)) == False:
                        move_airport = False
                        # print(name, 'missing', band)

                # If the airport has all bands move the files into new
folders
                if move_airport == True:
                    num_airports += 1
                    # Move images band by band
                    for band in band_labels:
                        filename = name + '_' + band + '.TIF'
                        oldFilePath = os.path.join(loadPath,
airport_type, entityID, filename)
                        newFilePath = os.path.join(savePath,
'airports', band, filename)
                        # Handle name conflicts for multiple images of
same airport
                        i = 0
                        while (fileExist(newFilePath)):
                            newFileName = name + '_' + band + '_' +
str(i) + '.TIF'
                            newFilePath = os.path.join(savePath,
'airports', band, newFileName)
                            i += 1
```

```
                        # Move the file
                        copyfile(oldFilePath, newFilePath)
    return num_airports

def non_airportsMove(loadPath, savePath, quantity):
    num_moved = 0
    selected_tiles = set()
    print('Moving', quantity, 'non_airports')
    # Make sure every on airport has an image in these bands
    band_labels = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8']

    # List entityIDs
    entityIDs = os.listdir(loadPath)

    # Selects one tile round-robin from each entityID until the num of
non_airports equals the num airports
    while 1:
        for entityID in entityIDs:
            if num_moved < quantity:
                imgs = os.listdir(os.path.join(loadPath, entityID))
                if len(imgs) != 0:
                    # Select random image from directory
                    rand_index = randint(0, len(imgs) - 1)
                    img = imgs[rand_index]
                    name = img.split('_')[0]

                    # Check if this tile was selected already
                    before_add = len(selected_tiles)
                    selected_tiles.add(os.path.join(entityID, img))
                    after_add = len(selected_tiles)

                    if before_add != after_add:
                        # This is a new tile
                        band_check = True
                        darkness_check = True
                        # Check if all bands exist
                        for band in band_labels:
                            filename = name + '_' + band + '.TIF'
                            if fileExist(os.path.join(loadPath,
entityID, filename)) == False:
                                band_check = False

                        if band_check == True:
                            # Test if any of the images are black
or grey
                            darkness_check =
checkTile(os.path.join(loadPath, entityID, filename))
                        else:
                            darkness_check = False

                    if darkness_check == True:
                        # Move all bands
                        for band in band_labels:
                            filename = name + '_' + band + '.TIF'
```

```
                                oldFilePath = os.path.join(loadPath,
entityID, filename)
                                newFilePath = os.path.join(savePath,
'non_airports', band, filename)
                                # Handle name conflicts for multiple
images of same airport
                                i = 0
                                while (fileExist(newFilePath)):
                                    newFileName = name + '_' + band +
'_' + str(i) + '.TIF'
                                    newFilePath =
os.path.join(savePath, 'non_airports', band, newFileName)
                                    i += 1
                                copyfile(oldFilePath, newFilePath)
                            num_moved += 1
            else:
                return

def consolidate(tilePath, savePath, genMissing=False):

    # Create the new file structure
    band_labels = ['B2', 'B3', 'B4', 'B5', 'B6', 'B7', 'B8']
    class_labels = ['airports', 'non_airports']

    createDir(savePath)
    for class_type in class_labels:
        createDir(os.path.join(savePath, class_type))
        for band in band_labels:
            createDir(os.path.join(savePath, class_type, band))

    # Move tiles into new folders & check for band consistency
    batches = os.listdir(tilePath)

    # Loop through each batch of tiles
    for batch in batches:
        num_airports_moved = airportsMove(os.path.join(tilePath, batch,
'airports'), savePath)
        print(batch, num_airports_moved)
        non_airportsMove(os.path.join(tilePath, batch, 'non_airports'),
savePath, num_airports_moved)

    return num_airports_moved
```

gen_npy_images.py

```
# Creates images from .TIF tiles and saves them as numpy arrays
# Splits the generated images up into training, validation, and testing
sets
# Creates dictionary files of image paths for use by datagenerator

import os
import numpy as np
from PIL import Image

# Creates a directory if one doesn't exist
```

```
def createDir(path):
    if not os.path.exists(path):
        os.makedirs(path)

# Adds band to filename
def bandName(filename, band):
    file_splt = filename.split('_')
    if len(file_splt) > 2:
        name = file_splt[0] + '_' + band + '_' + file_splt[2]
    else:
        name = file_splt[0] + '_' + band + '.TIF'
    return name

# Returns a name with band removed
def rawName(filename):
    file_splt = filename.split('_')
    if len(file_splt) > 2:
        tif_rmv = file_splt[2].split('.')
        name = file_splt[0] + '_' + tif_rmv[0]
    else:
        name = file_splt[0]
    return name

# Load up individual colors and combine into a composite image
(channels last)
def createImages(filenames, loadPath, savePath, colors, img_size):
    # Loop through each target/non_target
    for filename in filenames:
        # Initialize array in proper shape
        composite_img = np.empty([img_size, img_size, len(colors)],
dtype=np.uint16)
        for i, color in enumerate(colors):
            # Get band independent filename
            band_filename = bandName(filename, color)
            # Load TIF image
            img = Image.open(os.path.join(loadPath, color,
band_filename))
            composite_img[:, :, i] = img
        # Save composite image as numpy array
        raw_name = rawName(filename)
        np.save(os.path.join(savePath, raw_name), composite_img,
allow_pickle=True, fix_imports=True)
        print('Saved new', len(colors), 'color image at',
os.path.join(savePath, raw_name))

def createDataset(loadPath, savePath, colors, img_size=256, train=.6,
val=.2, test=.2, pickle=False):
    # Create file structure of dataset
    class_labels = ['airports', 'non_airports']

    # Top level dir
    createDir(savePath)

    # Create train/val/test dirs
    if train != 0.0:
```

```
        createDir(os.path.join(savePath, 'train'))
        for class_type in class_labels:
            createDir(os.path.join(savePath, 'train', class_type))

    if val != 0.0:
        createDir(os.path.join(savePath, 'val'))
        for class_type in class_labels:
            createDir(os.path.join(savePath, 'val', class_type))

    if test != 0.0:
        createDir(os.path.join(savePath, 'test'))
        for class_type in class_labels:
            createDir(os.path.join(savePath, 'test', class_type))

    # Determine the number of total images available & size of
train/val/test
    target_imgs = sorted(os.listdir(os.path.join(loadPath, 'airports',
'B2')))
    non_target_imgs = os.listdir(os.path.join(loadPath, 'non_airports',
'B2'))
    train_size = np.int(len(target_imgs) * train)
    val_size = np.int(len(target_imgs) * val)
    test_size = np.int(len(target_imgs) * test)

    # Get filenames for targets
    target_train = target_imgs[0:train_size]
    target_val = target_imgs[train_size:train_size + val_size]
    target_test = target_imgs[train_size + val_size:train_size +
val_size + test_size]

    # Get filenames for non_targets
    non_target_train = non_target_imgs[0:train_size]
    non_target_val = non_target_imgs[train_size:train_size + val_size]
    non_target_test = non_target_imgs[train_size + val_size:train_size
+ val_size + test_size]

    # Generate the Images
    createImages(target_train, os.path.join(loadPath, 'airports'),
                 os.path.join(savePath, 'train', 'airports'), colors,
img_size)
    createImages(target_val,  os.path.join(loadPath, 'airports'),
                 os.path.join(savePath, 'val', 'airports'), colors,
img_size)
    createImages(target_test,  os.path.join(loadPath, 'airports'),
                 os.path.join(savePath, 'test', 'airports'), colors,
img_size)

    createImages(non_target_train,  os.path.join(loadPath,
'non_airports'),
                 os.path.join(savePath, 'train', 'non_airports'),
colors, img_size)
    createImages(non_target_val, os.path.join(loadPath,
'non_airports'),
                 os.path.join(savePath, 'val', 'non_airports'), colors,
img_size)
```

```
    createImages(non_target_test, os.path.join(loadPath,
'non_airports'),
                 os.path.join(savePath, 'test', 'non_airports'),
colors, img_size)

    return None
```

dictionary_gen.py

```
# This script creates dictionary files which contain the path to an
image & it's class
import os
import pickle

def save_obj(obj, name ):
    with open(name + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

def load_obj(name):
    with open(name + '.pkl', 'rb') as f:
        return pickle.load(f)

def createDict(dataset_path):
    # Get training/val/test dirs
    data_subsets = [d for d in os.listdir(dataset_path) if
os.path.isdir(os.path.join(dataset_path, d))]

    for subset in data_subsets:
        # Create a dictionary file
        new_dict = {}
        classes = os.listdir(os.path.join(dataset_path, subset))
        for i, single_class in enumerate(classes):
            images = os.listdir(os.path.join(dataset_path, subset,
single_class))
            for image in images:
                image_path = os.path.join(dataset_path, subset,
single_class, image)
                new_dict[image_path] = i

        print(subset, 'dictionary created')
        save_obj(new_dict, os.path.join(dataset_path, subset +
'_dict'))
```

Bibliography

[1] Alberts, D. S., Garstka, J., & Stein, F. P. (1999). Network Centric Warfare Developing and Leveraging Information Superiority. C4ISR Cooperative Research Program (CCRP) Publication Series. https://doi.org/10.1109/EURCOM.2000.874819

[2] Enterprise Capability CollaborationTeam. (2016). Air Superiority 2030 Flight Plan, (May), 11. Retrieved from http://www.af.mil/Portals/1/documents/airpower/Air Superiority 2030 Flight Plan.pdf

[3] Joseph F. Dunford, Jr., USMC, F. M. P. (2016). Focus on Third Offset Strategy. Joint Force Quarterly, (82), 136.

[4] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems (pp. 1097–1105). Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[5] Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. International Conference on Learning Representations (ICRL), 1–14. https://doi.org/10.1016/j.infsof.2008.09.005

[6] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 1–13. https://doi.org/10.1007/978-3-319-24553-9

[7] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539

[8] Yuan, J. (2016). Automatic Building Extraction in Aerial Scenes Using Convolutional Networks. Retrieved from http://arxiv.org/abs/1602.06564

[9] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., … Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[10] Uijlings, J. R. R., Van De Sande, K. E. A., Gevers, T., & Smeulders, A. W. M. (2013). Selective search for object recognition. International Journal of Computer Vision, 104(2), 154–171. https://doi.org/10.1007/s11263-013-0620-5

[11] Barsi, J.A.; Lee, K.; Kvaran, G.; Markham, B.L.; Pedelty, J.A. The Spectral Response of the Landsat-8 Operational Land Imager. Remote Sens. 2014, 6, 10232-10251.

[12] R. Kemker and C. Kanan, "Self-Taught Feature Learning for Hyperspectral Image Classification," in IEEE Transactions on Geoscience and Remote Sensing, vol. 55, no. 5, pp. 2693-2705, May 2017.

[13] E. Maggiori, Y. Tarabalka, G. Charpiat and P. Alliez, "Convolutional Neural Networks for Large-Scale Remote-Sensing Image Classification," in IEEE Transactions on Geoscience and Remote Sensing, vol. 55, no. 2, pp. 645-657, Feb. 2017.

[14] L. Jiao, M. Liang, H. Chen, S. Yang, H. Liu and X. Cao, "Deep Fully Convolutional Network-Based Spatial Distribution Prediction for Hyperspectral Image Classification," in IEEE Transactions on Geoscience and Remote Sensing, vol. 55, no. 10, pp. 5585-5599, Oct. 2017.

[15] L. Mou, P. Ghamisi and X. X. Zhu, "Deep Recurrent Neural Networks for Hyperspectral Image Classification," in IEEE Transactions on Geoscience and Remote Sensing, vol. 55, no. 7, pp. 3639-3655, July 2017.

[16] DigitalGlobe. (2014). WorldView-3, 2. Retrieved from http://www.digitalglobe.com/sites/default/files/DG_WorldView3_DS_2014.pdf

[17] DigitalGlobe. (2016). WorldView-4, 2. Retrieved from https://dg-cms-uploads-production.s3.amazonaws.com/uploads/document/file/196/DG2017_WorldView-4_DS.pdf

[18] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386-408.

[19] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological cybernetics, 36(4):193–202, 1980.

[20] S. Geman, E. Bienenstock and R. Doursat, "Neural Networks and the Bias/Variance Dilemma," in Neural Computation, vol. 4, no. 1, pp. 1-58, Jan. 1992. doi: 10.1162/neco.1992.4.1.1

[21] James, G., et al.: An Introduction to Statistical Learning with Applications in R, 1st edn. Springer, New York (2013)

[22] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15, 1929–1958. https://doi.org/10.1214/12-AOS1000

[23] Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Arxiv, 1–11. https://doi.org/10.1007/s13398-014-0173-7.2

[24] S. J. Pan and Q. Yang, "A Survey on Transfer Learning," in IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345-1359, Oct. 2010.

[25] Basu, S., Ganguly, S., Mukhopadhyay, S., DiBiano, R., Karki, M., & Nemani, R. (2015). DeepSat - A Learning framework for Satellite Imagery, 1–22. https://doi.org/10.1145/2820783.2820816

[26] Homer, C.G., Dewitz, J.A., Yang, L., Jin, S., Danielson, P., Xian, G., Coulston, J., Herold, N.D., Wickham, J.D., Megown, K., 2015. (2011). Completion of the 2006 National Land Cover Database for the conterminous United States. Photogrammetric Engineering and Remote Sensing, 77, 858–566. https://doi.org/citeulike-article-id:4035881

[27] Www2.isprs.org. (2018). 2D Semantic Labeling - Vaihingen - ISPRS. [online] Available at: http://www2.isprs.org/commissions/comm3/wg4/2d-sem-label-vaihingen.html [Accessed 4 Jan. 2018].

[28] Www2.isprs.org. (2018). 2D Semantic Labeling - Potsdam - ISPRS. [online] Available at: http://www2.isprs.org/commissions/comm3/wg4/2d-sem-label-potsdam.html [Accessed 4 Jan. 2018].

[29] Roy, D. P., Kovalskyy, V., Zhang, H. K., Vermote, E. F., Yan, L., Kumar, S. S., & Egorov, A. (2016). Characterization of Landsat-7 to Landsat-8 reflective wavelength and normalized difference vegetation index continuity. Remote Sensing of Environment, 185, 57–70. https://doi.org/10.1016/j.rse.2015.12.024

[30] Slater J.A., Malys S. (1998) WGS 84 — Past, Present and Future. In: Brunner F.K. (eds) Advances in Positioning and Reference Frames. International Association of Geodesy Symposia, vol 118. Springer, Berlin, Heidelberg

[31] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., … Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[32] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. 2004. Editorial: special issue on learning from imbalanced data sets. SIGKDD Explor. Newsl. 6, 1 (June 2004), 1-6. DOI=http://dx.doi.org/10.1145/1007730.1007733

[33] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Arbor, A. (2014). Going Deeper with Convolutions. https://doi.org/10.1109/CVPR.2015.7298594

[34] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. Arxiv.Org, 7(3), 171–180. https://doi.org/10.3389/fpsyg.2013.00124

[35] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning Long Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks, 5(2), 157–166. https://doi.org/10.1109/72.279181

[36] Bottou L. (2010) Large-Scale Machine Learning with Stochastic Gradient Descent. In: Lechevallier Y., Saporta G. (eds) Proceedings of COMPSTAT'2010. Physica-Verlag HD

[37] Cormode, G., Karloff, H.J., & Wirth, A. (2010). Set cover algorithms for very large datasets. CIKM.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
22-03-2018	Master's Thesis	March 2016 – March 2018

TITLE AND SUBTITLE

Target Detection using Convolutional Neural Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Loibl, Robert P., Captain, USAF

5d. PROJECT NUMBER
18P334C

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-043

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory Space Vehicles Directorate
Kirtland AFB, New Mexico
(505) 846-1813 Robert.Sivilli.1@us.af.mil
ATTN: Robert M. Sivilli

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RVSV

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research explores the use of Convolutional Neural Networks (CNNs) to classify targets of interest within satellite imagery. Methods were specifically devised for the classification of airports within Landsat-8 scenes. A novel automated dataset generation technique was developed to create labeled datasets from satellite imagery using only coordinate metadata. Using this approach a very large dataset of over 132,000 labeled images was created without human input. This dataset was used to evaluate the effects of color and resolution on airport classification accuracy. Two experiments were run with the first experiment classifying large airports with 96.8% accuracy, and the second classifying large and medium airports with 90.2% accuracy. Additionally, a new algorithm was developed which optimizes the selection of multi-spectral color bands in order to best trade-off classification accuracy for the number of spectral bands employed.

15. SUBJECT TERMS

Artificial Intelligence, Neural Networks, Image Classification, Remote Sensing, Multi-Spectral

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Kenneth M. Hopkinson , AFIT/ENG
U	**U**	**U**	**UU**	93	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636 Kenneth.hopkinson@afit.edu

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18